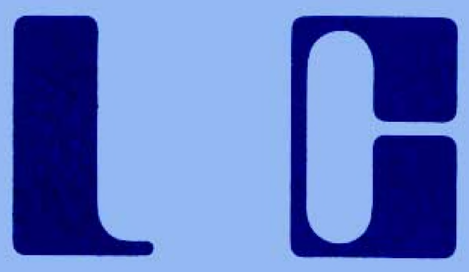


C COMPILER

by

MISOSYS

Copyright (C) 1982 by Jim Frimmel

```
LL          CCCCCCC
LL          CCCCCCC
LL          CC   CC
LL          CC
LL          CC   CC
LLLLLLL     CCCCCCC
LLLLLLL     CCCCCCC
```

C-Language Compiler

Reference Manual

Copyright (C) 1982, 1983 by Jim Frimmel
All rights reserved

Reproduction in any manner, electronic, mechanical, magnetic, optical, chemical, manual, or otherwise, without written permission, is prohibited.

Published by:
MISOSYS
P. O. Box 4848
Alexandria, Virginia 22303-0848

V 1.1 920687

LIMITED WARRANTY

MISOSYS shall have no liability or responsibility to the purchaser or any other person, company, or entity with respect to any liability, loss, or damage caused or alleged to have been caused by this product, including but not limited to any interruption of service, loss of business and anticipatory profits, or consequential damages resulting from the operation or use of this program.

Should this program recording or recording media prove to be defective in manufacture, labeling, or packaging, MISOSYS will replace the program upon return of the program package to MISOSYS within 90 days of the date of purchase. Except for this replacement policy, the sale or subsequent use of this program material is without warranty or liability.

WARNING

This program package is copyrighted with all rights reserved. The distribution and sale of this program is intended for the personal use of the original purchaser only and for use only on the computer system noted herein. Furthermore, copying, duplicating, selling, or otherwise distributing this product is expressly forbidden. In accepting this product, the purchaser recognizes and accepts this agreement.

ATTENTION

A limited licensing agreement is available for the generation of commercial products compiled with LC and using the LC run time library. Contact the publisher for details.

MISOSYS
P. O. Box 4848
Alexandria, Virginia 22303-0848
703-960-2998

LC is a trademark of MISOSYS.
LDOS is a trademark of Logical Systems, Inc.
TRS-80 and TRSDOS are trademarks of Tandy Corporation.
UNIX is a trademark of Bell Telephone Laboratories.

F O R W A R D

LC is an exciting product for the TRS-8Ø series of microcomputers. The power of the C language is beginning to be realized by many individuals. LC puts the power of C into your hands. The unique implementation of the LC/EDAS language development system provides most of the standard C language capabilities as described by Kernighan and Ritchie in "The C Programming Language" as well as an extensive macro assembler.

LC has been under development for two years - going through various transitions of implementation. The product you now own, has many unique features. It will provide you with extreme fascination of new techniques in coding. You not only have the opportunity of learning a relatively new compiled language, but you also have the opportunity to instill advanced programming techniques into your programs. You will get out of LC much more than what you give it. If you are new to the C language, take the time to learn it. You will be greatly rewarded.

The tenure of the LC project has been intriguing. That one word certainly does not sum up the entire history, however. No one word can. Frustrating, rewarding, despair, hope, and many other words can be added to the list. During LC's long labor, Jim's wife Sam, had some of her own, giving birth to a new daughter, Danielle. The encouragement and support that Jim's family provided him shall not go unnoticed - let alone the deep understanding of all those late hours at the computer.

A special thanks goes out to Rich Deglin for his continued support throughout LC's design and implementation life cycle. Rich is also remembered for his contributions to the installation library of the string and control functions and his many suggestions of enhancements to the standard library. I'd like to "point" out that Karl Hessinger's plotting functions will be used by many LC programmers and will be most appreciated.

To Jim Frimmel goes unending appreciation for a job well done. This astute mind has formed many of the sophisticated techniques inherent in LC. It has been my personal pleasure to have worked so closely with Jim over the long LC development time frame. It will be equally pleasurable to continue this association. Jim's efforts were sparked by the work done on the SMALL C compiler by Ron Cain - one who has certainly provided the foundation for a lot of C work.

Lastly, let me acknowledge the understanding that my wife, Brenda, has demonstrated. Those long hours spent in putting this reference manual together were time apart from her. For a newly married couple, it was a noble sacrifice. It will not be forgotten.

Roy Soltoff

Publisher

FROM THE AUTHOR

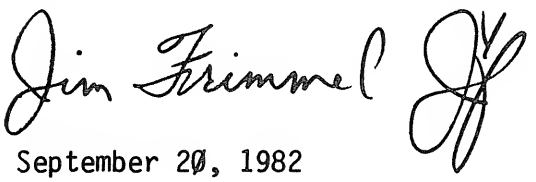
The product which you have in your hands, affectionately called "Elsie", has been a labor of love. She has grown with my children, who have shared in this labor by giving up time with their father. My wife, Sam, has nourished Elsie with her understanding, kindness and love, and has sacrificed her time as well. I have long awaited the day when Elsie would be completed, so that I could share more with them. This family, whom I love dearly, is my greatest earthly treasure, beyond price. My Heavenly Father's love and patience, and the gifts he has blessed me with, are what made it all possible. I pray that you will share in my blessings in some way, through whatever Elsie brings your way.

Elsie has grown from a "small seed" of generosity: Ron Cain's Small-C compiler. Ron published his compiler and placed it in the public domain for all to hack, and many have done so. Ron Cain deserves our heartfelt thanks for stirring up interest in the C language and for getting us started. Elsie was "bootstrapped" using Small-C, and she shows her roots in places. It is my hope that Elsie will bring the C language to a less experienced audience than that which the language now appeals to. I am sure there are many who, with just a little help getting started with C, will become excellent C'ers.

Elsie would not be here if it had not been for Roy Soltoff of MISOSYS. He has been a generous benefactor and friend, giving of his time, hardware and software to bring Elsie to you. His artist's eye also gave this manual the friendly feel and utility that his products are known for. What makes the Elsie package especially unique is the work that Roy put into the EDAS IV assembler. This assembler is the best example of "user-driven" software that I know of. Roy listens to his users (and responds), and he has gotten quite an earful from me. Most of all, he has been more patient than I could have imagined, waiting for my perfectionist dreams to become useful reality. Roy's bride, Brenda, has displayed this same patience and kindness, with nary a protest. I'm sure many others are with me when I say, thank you and may God bless you both.

Elsie has been helped along the way to becoming a product by some good friends. Karl Hessinger and Rich Deglin were especially generous with their time and programming skills. Steve Hill and Scott Loomer also helped with suggestions, advice and feedback. Also, my wife Sam's editorial skills have resulted in an unusual combination: sound grammar in a technical manual. To all of you who have helped Elsie along, thank you and may God bless you.

To my new daughter, Danielle Michelle, who, like Elsie, is full of bright promise and laughter.



September 20, 1982

P R E F A C E

Although considerable effort was expended to make the LC reference manual as complete as possible, this documentation package in no way is to be considered an instructive guide into the writing of C language source programs. Chapter two is a good definition of the C language as implemented in LC. Although some may find that complete enough, one reference text is available that must be added to your library. "The C Programming Language", by Brian W. Kernighan and Dennis M. Ritchie is the "bible" of C and is MUST reading. It is filled with numerous examples and illustrations of each C statement and also contains example upon example of useful functions. If you are new to assembly language, the preface contained in the EDAS reference manual should be consulted for additional information.

The advice is to peruse the contents of both this LC reference manual and the EDAS reference manual to familiarize yourself with its information and content. If you have any questions concerning the LC development system, feel free to call or write; however, since the results of a C program rely heavily on exact syntax, if your question concerns any aspect of the C language, it must be submitted in writing. All transactions need to be identified with your registration number(s) so be prepared to provide both your LC and EDAS registration numbers. It would also be helpful to make sure your questions are not answered in the manual.

Speaking of registrations, MISOSYS would like to provide you with the best technical support possible. To provide this support, we need to know who our customers are. So please fill out the registration form packaged with the diskette and return it to us promptly - postal card postage is sufficient. The registration number located on the diskette labels must be entered onto the registration card and should also be entered in the space provided below. The registration number must be mentioned on all correspondance with us or when telephoning for service, so don't lose it.

Registrations: LC 920687 EDAS 821040

<<**>>
<<**>>
<<****** MISOSYS LC - C Language Compiler *****>>
<<****** Copyright 1982, 1983, by Jim Frimmel *****>>
<<**>>
<<**>>

Table of Contents

Forward i
From the Author ii
Preface iii
Introduction 1-1
  Elsie Files 1-3
  LC Environment 1-5
  Standard Input/Output 1-6
  Standard I/O Redirection 1-7
  Standard Header Files 1-8
  Function Libraries 1-9
  Closing Comments 1-10
Language Definition 2-1
  Program Environment - Functions 2-1
  Statements - Simple & Compound 2-3
  Data Representation - Constants 2-4
  Variable Names (Identifiers) 2-6
  Data Declarations 2-8
  Scope of Variables & Functions 2-9
  Storage Classes 2-10
  Expressions 2-13
  Unary Operators 2-17
  Binary Operators 2-19
  Statements 2-24
    BREAK 2-28
    CONTINUE 2-29
    DO-WHILE 2-27
    FOR 2-27
    GOTO 2-30
    IF 2-25
    RETURN 2-29
    SWITCH-CASE-DEFAULT 2-26
    WHILE 2-27
LC Operators Guide 3-1
  LC Operation 3-1
  File Specifications 3-3
  Compiler Switch Options 3-3
  Creating a CMD File 3-5
  Compiler Directives 3-7
  A Simple Exercise 3-10
LC Library 4-1
  Standard Library (LC/LIB) 4-2
  Installation Library (IN/LIB) 4-22
    Primitive Plotting Functions 4-23
    Advanced Plotting Functions 4-25
    Plotting Control Functions 4-27
    String Functions 4-29

Third Edition

Control Functions 4-34
Floating Point Library (FP/LIB) 4-37
Single Precision Operations 4-40
Double Precision Operations 4-42
Data Conversion Functions 4-43
Advanced Topics 5-1
Utilizing Assembly-Time Options 5-1
ARGS 5-1
ERRORMSG 5-1
FIXBUFS 5-1
FPLIB 5-1
GETNL 5-2
INLIB 5-2
KBECHO 5-2
MAXFILES 5-2
REDIRECT 5-2
ZVAR 5-2
Separate Compilation 5-3
Using the -Global Option 5-3
Using Extern and Static 5-3
Assembling Separately Compiled Modules 5-4
Creating User Libraries 5-5
In-Line Libraries 5-5
PDS Libraries 5-6
Assembly Language Output Structure 5-8
Program Memory Map 5-8
LC Macros 5-9
LC Identifier Output 5-11
Run-Time Subroutines 5-12
Assembly Language Interfacing 5-13
Register Usage 5-13
Argument Passing 5-13
Labels and Constants 5-14
When Things Go Wrong 5-16
Compilation Errors 5-16
Assembly Errors 5-17
Appendices ..... ABC
Operators A-1
Library Functions B-1
Compiler Error Messages C-1
Sample Programs D-1

L E T ' S M E E T E L S I E

Before diving into Elsie, we suggest that you do the following, at a minimum:

1> Read the entire introduction to get an idea of what LC is all about.

2> Make backup copies of the distribution diskettes. Model I/III Elsie is released on two distribution diskettes. The diskette labeled "LC", contains the LC compiler and all support libraries. The second diskette labeled "EDAS", contains the macro assembler and editor and various EDAS support utilities. PRO-LC includes the compiler and macro-assembler on one double density diskette. We suggest that you make one set of archival backups and store them away in a secure area (safe from dust, dirt, magnetic fields, etc.). Then make a working backup of the distribution diskettes. The procedures for making backup copies can be located in the UTILITY section of your DOS user manual under "BACKUP". For your information, LC is distributed on a data diskette(s).

3> Create a DOS system diskette with a maximum of free space. If you are going to BOOT from this diskette, then you will need a configuration that includes the DOS keyboard driver (Model I/III) at the barest minimum. Also, if you are using a Model I, use the CDO/FLT filter in your configuration to filter the video driver. The filter presents more usable character indications for the characters, "[", "\", "]", "^", and "~". This "working system diskette" can be created by using the DOS PURGE utility on a fresh backup of LDOS. You may remove all files except SYS0-SYS4, SYS6, SYS8 (SYS8 can be removed from TRSDOS 6.x), SYS10-SYS12. Keep any other file you use frequently (for instance, BACKUP and FORMAT). If you are using a double density system, follow steps <3A>. If you are using a single density system, follow steps <3B>.

3A> A 40-track double density minimal system diskette per the above has about 144K free. Copy EDAS/CMD from the working EDAS diskette. Then copy all of the files from the working LC backup to this LC system diskette. This should still leave some work space. Use a data diskette in your second drive for LC source files and all files output by LC and EDAS.

3B> A 35-track single density minimal system diskette per the above has about 57K free. Copy EDAS/CMD from the working EDAS diskette, then copy LC/CMD, LC/JCL, LC/ASM, LCMACS/ASM, and STDIO/CSH from the working LC backup to the new system diskette. We'll call this diskette your LC System diskette. Now remove the LC/CMD, LC/JCL, LC/ASM, LCMACS/ASM, and STDIO/CSH files from the working LC backup (the one you just copied FROM - not the system disk that just received those files). Re-designate this disk your working LC Data disk. In a two drive system, your work files (LC source and output files) will need to be stored on this LC Data disk. Therefore, make a few backups of this disk to use for various LC programming sessions.

4> Notice that LC requires a two drive system. If you have a one-drive system, you overlooked the machine requirements noted in the LC advertisements and in the catalog. We would not want any user so frustrated with trying to utilize the LC compiler system on a single-drive computer.

5> If you are really anxious to proceed and create a working C-language program, turn to the OPERATOR'S GUIDE page 3-10 and note the "simple exercise". Place the LC system diskette in drive 0, the LC data diskette in drive 1, and proceed with the EXERCISE.

6> Read the rest of the LC user's manual. Pay particular attention to the OPERATOR'S GUIDE and the LANGUAGE DEFINITION chapters. Skimming is OK for technical details, but get a feel for where things can be found in your LC manual.

7> Have fun with Elsie!

ELSIE FILES

Elsie comes complete with everything you will need to turn your LC source programs into executable CMD programs. There are many files on your distribution diskettes: a compiler, an editor/assembler, utilities, libraries, and JCL files. Here is a description of some of these files and their uses:

LC/CMD

This is the LC language compiler. LC accepts as input, C source code files, and outputs an EDAS Version IV compatible assembly source file. In order to organize files in a structured manner, LC source code files have a file extension of "/CCC" and LC output assembler files have a file extension of "/ASM".

LC/LIB

This is the standard function library. It is an implementation of the portable library available under most installations of C compilers. These functions allow programs to be written which will be directly usable under other C language systems that have the standard library available. The standard functions perform such tasks as input/output, dynamic memory allocation, standard I/O redirection, and string handling. The standard library was designed to be compatible with the standard library under Western Electric's UNIX operating system.

FP/LIB

This contains the LC floating point function library. LC does not have floating point variables built into the language itself. However, this library supplies access to the floating point routines in the TRS-80 ROM through functions. Single and double precision arithmetic, as well as trigonometric and transcendental functions, are included. FP/LIB also includes functions for converting between ASCII, integer, single and double precision variables.

IN/LIB

The installation function library is contained in this module. These functions supply graphics plotting, string manipulation, certain LDOS entry points, and other TRS-80 features.

STDIO/CSH

The standard I/O header file supplies constants and definitions which are needed to use the standard I/O library (LC/LIB).

LC/ASM

This file is the primary assembler source file assembled by EDAS. It accesses your "main" program and establishes the neccessary interfacing between your program and the LC runtime modules needed to make a "complete" CMD program.

LCMACS/ASM

This file contains the #option defaults and assembly language macros referred to by the compiler and used by the assembler. This file is always automatically accessed by EDAS when assembling the LC/ASM file.

CDO/FLT

This file is supplied with the Model I/III release of LC. It is a video filter for the Model I machine. It provides a more reasonable and less confusing display of the characters: [, \,], ^, ~.

LC ENVIRONMENT

LC was designed to be compatible with C programs written and intended to run under UNIX. Thus some features of UNIX were incorporated into the design. These features include standard I/O devices, standard I/O redirection, device independence, command line arguments, and dynamic memory allocation.

To make C a portable language, the interface within a program to the external world is isolated in a standard library. A program written in C using only the standard functions to perform input, output and memory allocation can be transported in source code form to another sytem, recompiled, and run with minimal changes. The LC system includes a standard library which is compatible with programs developed under UNIX. Thus, programs developed under LDOS with LC will run under UNIX as well. The reverse is also true, except in cases where features not implemented in LC are used in the program developed under UNIX.

LDOS runs on a variety of machines, with many varied configurations of I/O devices. Any program written for one particular environment running LDOS can run on any other, provided that the standard library is utilized. The programmer should bear this in mind when writing programs to avoid conversions when migration to a new system becomes necessary.

STANDARD INPUT/OUTPUT

Any program generated by LC will normally have three files automatically opened when the program begins execution. These files are standard input, "stdin" (normally the console keyboard); standard output, "stdout" (normally the console display); and standard error, "stderr" (normally the console display). The program can access these files without opening them by using standard library functions since the LC standard library automatically opens these standard files. They are also automatically closed when the program is exited as well. Thus, the program which uses the standard I/O files exclusively can deal with input and output and leave the opening and closing to the LC standard library.

STANDARD I/O REDIRECTION

The standard I/O files normally operate to and from the user's console. However, a facility is inherent within the LC standard library to permit you to "re-direct" any of the standard I/O devices - thus the term "I/O redirection". The user can give a file specification that will be used in place of the normal specification when a standard file is opened. This is done on the LDOS command line when the user executes the program.

When the left angle bracket symbol, "<", appears on the command line, followed by a file specification, that file specification is used when the standard input file is opened. Similarly, the right angle bracket symbol, ">", causes substitution of the standard output file specification, the ">>" causes standard output to be appended to the redirected file/device, and the number sign symbol, "#", causes substitution of the standard error file specification. Spaces are NOT permitted between the redirection character and the file specification.

It may not be immediately obvious how this feature can be used. Here is an example LC program that illustrates the straightforward use of standard I/O redirection. The following program can be used to copy any file to any other file (remember that "file" can be any device or LDOS disk file).

```
/*  CLONE  -  copy standard input to standard output  */
#include stdio/ccc
int c;
main ()
{    while ((c = getchar()) != EOF) putchar(c);    }
```

The example program simply copies the standard input to the standard output until end of file is reached. Once this program is compiled and assembled it can be used to copy any file to any other. For example:

```
CLONE <CLONE/CCC
```

will display the file clone/ccc on the system console. The command:

```
CLONE >*PR
```

lets the user type to the system printer. If disk file copying is needed, the command:

```
CLONE <INFILE/ASM:1 >OUTFILE/BAK:2
```

will copy the file "INFILE/ASM:1" to the file "OUTFILE/BAK:2". If the user wishes to have a printed log of any error messages that a program puts out, use something like :

```
LC TESTLIB #*PR
```

Any messages that LC outputs to the standard error file will be re-directed to the printer device in lieu of the console display.

STANDARD HEADER FILES

Standard header files are files which contain definitions peculiar to a system. They usually take the form of "#define" statements and "extern" statements within the header file. In order to use certain libraries, a corresponding header file should be included (using the "#include" statement). The file extension of "CSH" is used for "LC Standard Header" files.

A program to be compiled and linked with LC should have the file "STDIO/CSH" included to compile properly. STDIO/CSH also defines various system dependent parameters, such as end of file (eof) and end of line (eol). <stdout>, <stdin>, and <stderr> are addresses in the standard library which do not need to be defined before use.

The following listing is representative of the STDIO/CSH file included with the LC package.

```
/*  ELSIE STANDARD I/O CONSTANTS  */
#define EOF      -1
#define eof      -1
#define eol      13
#define EOL      13
#define FILE     char
#define NULL     Ø
#define TRUE     1
#define FALSE    Ø
#define true     1
#define false    Ø
#define OFF      Ø
#define ON      -1
```

FUNCTION LIBRARIES

Commonly used LC functions are collected into FUNCTION LIBRARIES. The functions in a library can be used by the programmer without the need to rewrite or recompile the functions needed. Once an LC program has been compiled, it can then be linked during the assembly phase with the functions it requires. Only those functions necessary for the execution of the program are linked to the compiled program.

Certain functions required by many programs are included in a special library called the STANDARD LIBRARY. The standard library is the common denominator between all C language installations. Programs written using functions in the standard library are easily transported to any other computer supporting a C language system with the standard library implemented. The most important aspect of the standard library is that it allows the details of each system's peculiar operating environment to be hidden from the programmer's view. The standard library provides the functions for input/output, memory allocation, and character set manipulations. In addition, a collection of subroutines used by the compiled C program to perform basic operations is also supplied in the standard library.

Users can also create their own collections of often-used functions that can be used in the same manner as the standard library. These USER LIBRARIES reduce the programming time, compilation time, and program complexity necessary in creating new programs. Functions, once defined, written, and tested, can be added to the user library and need only be referred to by name in later programs. The linking process brings the functions into subsequent programs without the need to recompile. If you want to create and maintain your own libraries, you will need the Partitioned Data Set (PDS) utility.

Special purpose libraries may also be created for use in particular types of applications. For instance, the functions specific to the TRS-80 are provided with your LC package are in the special purpose library, IN/LIB. This is an example of how the C language avoids the trap of non-standard extensions being included within the language.

CLOSING COMMENTS

LC encourages the use of structured programming methods. Unless one uses the "goto" statement heavily, LC practically demands a structured approach to program construction. This is not to say that writing programs with LC will automatically make you a good, structured programmer. This is a skill that is developed by learning and applying the basics.

Some understanding of structured design concepts is necessary in order to effectively use LC. Probably the first frustrating thing that novice LC programmers will encounter, especially if their experience is limited to BASIC and assembly language, is the discouragement of the use of "goto". Kernighan and Ritchie, in THE C PROGRAMMING LANGUAGE, state that the "goto" is never necessary, and in practice it is almost always easy to write code without it. The concept to understand is that the "goto's" are hidden within the program statements. LC provides, in a coherent, understandable form, the program constructs that you have been building out of "goto's".

Last but not least, several texts are available that should be part of your library. The first, THE C PROGRAMMING LANGUAGE by Brian W. Kernighan and Dennis M. Ritchie (published by Prentice-Hall), is the Bible of the C language and is a required part of your own library. We will refer to this book throughout this manual by the abbreviation, "K&R", for Kernighan and Ritchie. Two other books, ELEMENTS OF PROGRAMMING STYLE and SOFTWARE TOOLS, by Brian W. Kernighan and P. J. Plauger (published by Addison-Wesley), present a good foundation of structured programming concepts. Prentice-Hall also publishes THE C PUZZLE BOOK by Alan R. Feuer. This book can be used to test your understanding of the C language. It has proven itself to be quite useful in testing out the intricacies of the LC compiler.

PROGRAM ENVIRONMENT - Functions

The C language is, in a word, functional. The basic unit of program construction when using LC is the function. Every LC program is a collection of functions. Each function is a collection of statements that work together to achieve (hopefully) a useful, well-defined, purpose.

Each function can have information passed to it when it is invoked ("called"). The elements of information passed to the called function are denoted as arguments. In LC, arguments are copied onto the stack. The function can then access and use the "local" (known only to the called function) arguments, leaving the original copy of the arguments unchanged. Each argument is defined at the start of the function. Functions also return values to the functions that call them. In LC this value is always a 16-it number. The value returned can be compared to, placed in a variable, etc. Functions can appear in an arithmetic expression anywhere that a constant can.

Here is an example of a function:

```
square(num)
   int  num;
{   return num * num ;  }
```

The function, square(), returns the square of a number; in other words, the argument, "num", is multiplied by itself and the result is returned. Arguments are listed in parentheses after the name of the function, separated by commas. These arguments must be passed by the calling function in the same order as they appear in this list.

The BODY of the function is the group of executable statements that are within the braces "{" and "}". Actually, the grouping of statements in between braces denotes a special kind of statement called the COMPOUND statement. The compound statement is fully explained in the section on LC language statements.

Every LC program has a special function called "main" which is always the entry point to the program. When referencing a function within this narrative, we will put "()" after the name to identify it as a function. This is close to the way it looks in an LC program. The function, main(), calls other functions, which in turn call other functions, etc... Thus, each program is an hierarchical structure of functions, with main() at the top of the hierarchy.

The LDOS command line which invokes the LC program is passed to the function main() using two parameters, "argc" and "argv". One LC program can invoke another program by using the cmd() function. When the called program finishes, a special function, exit(), is used to return a value to the calling program. Programs can call other programs, passing any arguments using "argc" and "argv". In a way, each program appears as a function to other LC programs and to LDOS.

Please scrutinize the illustration of functions in the following example:

```
main()
        {   /*  The "main" function ...
                execution begins here!
                */
            say_hello();
            do_work();
            say_goodbye();
            exit(Ø);  /* a normal exit, no error code */
        }   /* sorry, we can't "goto" any of the functions below. */

        say_hello()
        {   puts("Hiya!!!");    putchar(eol);   }

        say_goodbye()
        {   puts("Bye y'all!!!");   putchar(eol);   }

        do_work()
        {   while (not_quitting_time)
            {   attach(nut,bolt);
                pass_on(widget);
            }
        }
```

STATEMENTS - SIMPLE & COMPOUND

To create an LC function you have to state the action to be taken, using LC language STATEMENTS in the desired combination. Certain special statements are built into the language to provide the necessary programming constructs (sequence, iteration, selection). You may be surprised, at first, by the limited number of statements built into the C language. The authors of the language wished to maintain the generality of the programming statements, forcing any special features to be outside of the programming language itself. Other languages often have extensions in the form of statements to provide specialized features, leading to incompatible versions of the same language. BASIC is a well-known example of a language extended in far too many different ways. The C language avoids this situation by only providing those statements necessary for structuring the program's logical flow and by placing all special features into function LIBRARIES. Function libraries are nothing more than collections of commonly used functions. See the section on the LC libraries in the INTRODUCTION for more information.

Simple LC statements always end with a semicolon ";", the STATEMENT TERMINATOR. The LC compiler depends on the semicolon to tell when a simple statement ends. Any number of simple statements may be entered, one after the other, to form a SEQUENCE of statements that are executed one at a time, first to last.

The brace characters, "{" and "}", are used to enclose a sequence of statements to form a COMPOUND statement. A compound statement can be used anywhere a simple statement can be used. Thus, the body of a function (that portion enclosed in braces) is just a special form of compound statement.

For example:

```
n1 = Ø;
```

is a simple statement. However, the statement:

```
{
h = h / 2; xØ = xØ + h / 2; yØ = yØ + h / 2;
x = xØ + i * 32; y = yØ + 1Ø; u = x; v = y;
++i; p( 1, i );
}
```

is a compound statement.

DATA REPRESENTATION - CONSTANTS

Numbers and characters must be entered in your LC program in certain ways in order for the compiler to understand them properly. A fixed value to be used in an LC expression is called a CONSTANT.

Where just a decimal number is required, you can enter it just as you write it. A leading zero indicates that the constant is in another base. A leading zero followed by a string of digits indicates an OCTAL CONSTANT. A leading zero followed by 'X' or 'x' indicates that a hexadecimal constant follows. Thus, the decimal number, 255, can be represented as Ø377 or ØxFF, as desired.

If the variable to be assigned the constant is not big enough to contain the constant, only the least significant bits (LSB) of the number are stored. This is, in effect, storing the remainder of dividing the constant by 256 or 65,536, depending on the variable size. No warning is given when this happens, so the programmer must be sure that the variable can hold the number.

CHARACTER CONSTANTS supply a way to specify the code for a character which does not depend on any particular character set. A character constant is a list of characters within single quotes (apostrophes). For instance, the character constant 'A' is stored in the computer as the number 65 (in decimal). Again, it is up to the programmer to assure that the number of characters between apostrophes can fit into the variable being assigned. If more characters are specified than can fit, only the last one or two (as needed) are used.

When a sequence of characters is needed, a STRING can be specified by enclosing the characters between quotes (sometimes called "double" quotes - i.e. "This is a string"). LC does not place all of these characters into a variable but rather the ADDRESS of the first character of the string. Thus, when the string, "testing, 1 2 3", is used in an LC program, the characters between quotes are stored in memory, and the address of the first 't' is used in the expression where the string was specified. You can say that the number generated by LC to represent the string really POINTS to the string. The subject of POINTER variables, which are handy for manipulating strings, will be discussed later.

There are certain control characters that are needed frequently in programs, but which differ from machine to machine. These can be represented in C programs using ESCAPE SEQUENCES, to provide a machine-independent constant. The backslash character, "\", is called the ESCAPE CHARACTER and denotes the beginning of an escape sequence. A letter following the escape character indicates which control code is being specified. Also, certain characters that would otherwise be difficult to represent in strings and character constants are generated by following the backslash with the character. These escape sequences are shown in the following table:

Escape Sequence	Control Code	ASCII code used by LC	
\n,\N	NEWLINE character	x'0D'	CR
\t,\T	horizontal tab	x'09'	HT
\b,\B	backspace	x'08'	BS
\r,\R	carriage return	x'0D'	CR
\f,\F	form feed	x'0C'	FF
\\	backslash	x'5C'	backslash
\'	single quote	x'27'	apostrophe
\0	null	x'00'	null byte
\"	double quote	X'22'	double quote

In addition, any binary code can be represented in a string or character constant by following the backslash with a numeric constant. This is done by following the backslash with up to three octal digits. An extension which is not normally allowed in the C language is offered in the LC language as a convenience to microcomputer users who are only familiar with hexadecimal. The backslash may be followed by an 'x' and one or two hexadecimal digits. Either of these two methods result in an 8-bit character constant.

For example, the character 'A' can be represented as '\x41' using a hexadecimal escape sequence, or as '\101' in an octal constant. Similarly, to place a carriage return at the end of a line, the following three methods could be used:

```
"An example of a normal escape: \n"
"An example of a hexadecimal escape: \x0D"
"An example of an octal escape: \015"
```

When a character escape sequence is used within a string, the actual value of the escape sequence is stored in a string (i.e., only one byte of data per escape). Thus, the string:

```
"\n\x0d\015"
```

is only three bytes long in memory once the program is compiled and assembled.

VARIABLE NAMES

The names given to identify variables, functions, macros, and labels are called "identifiers" and all follow the same rules as to their format. LC identifiers may be of any length (be practical) and must start with an alphabetic character {'A' through 'Z', 'a' through 'z'} with the rest of the characters in the name consisting of upper-case or lower-case alphabetic characters {'A' through 'Z', 'a' through 'z'}, numeric characters {Ø through 9}, or the underline character {_}. LC will accept an underline as the first character of an identifier, however EDAS will not; therefore, do not start an identifier with the underline character.

LC remembers only the first eight (8) characters of an identifier, so these first eight must be unique.

Elsie is case-sensitive, i.e., recognizes the difference between lower-case and upper-case in identifiers. Thus, "EOF", "eof", and "Eof" are all different identifiers to Elsie. However, identifiers which must be written out in assembler source code for EDAS are converted to upper-case, since EDAS does not allow lower case assembly language code. A good, simple rule to follow is to use UPPER-case for macro constants only, and lower-case for all other identifiers. Since macro identifiers are not written to the assembly output file, they will not conflict with any other identifiers which are the same, except for case differences.

DATA DECLARATIONS

LC variables must always be declared before use. The standard procedure is to declare variables at the beginning of the program (globals) and at the beginning of each function (locals).

Character variables are stored in eight bits, or a byte. The declaration:

```
char c, string[81];
```

establishes a character variable named "c" and a character array named "string". Arrays of one dimension are allowed.

All other variable types (short, long, int, unsigned), as well as pointers, are stored in sixteen bits. The short and long declarations are provided in the interests of portability. The declarations:

```
int a;
short b;    short int b2;
long c;      long int c2;
```

are all acceptable declarations, and all result in the same size integer field. This is acceptable, since the C language does not guarantee that a "long" will be longer, or that a "short" will be shorter than integers. Integers declared in this manner are signed, i.e., their most significant bit is regarded as a sign bit. Their values can range from -32,768 to 32,767 (decimal). Unsigned fields do not have a sign bit. They range from Ø to 65,535 (decimal) and are declared like this:

```
unsigned  u;
unsigned int u2;
```

Arrays of one dimension are allowed for short, long, int, and unsigned types.

Pointer variables are different from the types described so far, in that they normally contain the ADDRESS of a data item. For example,

```
char *cp;
```

declares a pointer variable named "cp". The asterisk denotes INDIRECTION, i.e., that the data item is referred to indirectly through the pointer variable "cp". The address of the data item must be stored in the variable, "cp", before it is used as a pointer to access a data item. To refer to the data itself, an asterisk is placed before the name, e.g., *cp denotes the data item. An example of practical use follows:

```
getit(cp)
     char *cp;
{    while ((c=getchar()) != eol && c != EOF)
     {    *cp = c;
          ++cp;
     }
     *cp=NULL;
     return c;
}
```

The function, getit(), inputs characters continually from the standard input until end-of-file or end-of-line characters are encountered. When getit() is called, the pointer argument, cp, contains the address of a buffer area. One by one the characters are placed in the buffer, (*cp = c), and the buffer pointer is incremented (++cp).

Pointers may be declared for any data type. An alternative way of declaring a pointer is to leave out the size in an array declaration. For example,

```
int  count[];
```

declares an integer pointer, "prime". There is good reason for this method of declaring a pointer. Pointers may be INDEXED to get to the "nth" item in an array. Using the example above, count would contain the address of the beginning of an array of short integers. "count[Ø]" denotes the first element in the array, and "count[22]" denotes the 23rd element.

No matter how a pointer is declared, either method of using the pointer may be used as the programmer sees fit. Thus, "*count" and "count[Ø]" refer to the same data item and may be used interchangeably in the same program. Using "*count" is a little more efficient, however.

Pointers may point to other pointers. This bombshell of a statement is probably too much for you after the last few paragraphs; it must be said, however. LC allows pointers to have more than one LEVEL OF INDIRECTION. This can be declared several ways:

```
shine()
{    char *names[];
     char *(*words);
. . . . . .
```

Both of these declarations result in the same effect: a pointer which points to a pointer which points to a character field.

Pointer variables may have up to 32 levels of indirection. However, the practical limit is the ability of the programmer to keep track of all this. In general, two levels of indirection are all most folks can take.

More information and examples can be found in the reference text, "The C Programming Language", by Kernighan & Ritchie.

SCOPE OF VARIABLES & FUNCTIONS

Variables or functions which are declared outside of any function, i.e, are not parameters to functions or declared with braces, are called "external". They are external to all functions. External variables and functions can be used from any of the functions within the module being compiled. Using the "extern" statement, an external variable or function may even be accessed from another, separately compiled, module. Please do not confuse "extern" and external. External (to all functions) variables and functions can be declared without the "extern" statement. The "extern" statement is explained in full detail below.

Variables declared within a function are called "local". Functions may not be defined within another function, as is the case with the Pascal language. However, a function may be DECLARED "extern" so that it may be accessed within the currently defined function. Local variables may not be accessed from any other functions. They only exist for the function in which they are declared. Even within the function, a local variable can only be accessed in the block in which it is declared. Remember, a block is a section of code contained within a matching pair of braces.

Local variables can have the same name as external variables, or local variables declared in different blocks. If a local variable has the same name as an external variable then the local variable is the one accessed when used within the local block. In the following example:

```
int  same;
     funk(same)
     {    return same;   }  /* return local copy */
     hunk()
     {    if (block_1)
          {    int same;
               /*  some code could go here  */
          }
          else
          {    char same;
               /* some other code here */
          }
     }
```

every declaration of "same" was a unique variable. Although legal, the declaration of local variables with the same name within the same function is not recommended. This type of trickery, as shown in hunk(), needlessly causes confusion and is easily avoided.

STORAGE CLASSES

Variables and functions may be declared as being in certain classes. These classes specify where variables are to be stored. The classes available in LC are: auto, static, extern, and register. The storage class of an object is specified by placing the class name in front of the normal declaration:

```
auto char c;
static int ai[2Ø];
```

STORAGE CLASS - AUTO

Variables which are declared "auto" are stored on the stack. This is the default for variables declared within a function, so the declaration may therefore be omitted. Local variables which are "auto" are created afresh each time the function in which they are declared is called. This allows functions to be reentrant and recursive. Functions may not be declared with class "auto" since a function must be declared outside of any other function. As K&R say, the C compiler is incapable of compiling code onto the stack!

The scope of an auto variable is the block (within braces) in which it is declared. All other portions of the code being compiled are oblivious to the existence of the auto variable, and in fact there may exist other variables with the same name.

The auto class is illegal for functions and other external definitions (any variables declared outside of a function).

STORAGE CLASS - REGISTER

Variables declared in the register class are regarded as auto variables by LC, since the Z-8Ø has no extra registers available for use as register variables. Register variables are stored on the stack in the same manner and are also illegal outside of a function.

The scope of register variables is the same as that for auto variables.

STORAGE CLASS - EXTERN

The "extern" storage class allows an external variable declared in one module to be accessed from another module. A "module" is what is processed by one execution of LC, i.e., one set of C source input. Let's say that the following declaration:

```
int choice;
```

exists in module 1. If module 2 functions need to access this same variable, the declaration:

```
extern int choice;
```

would allow the access needed. LC will not reserve any storage for "choice" in module 2, since the storage class, "extern", tells LC that storage has been reserved in another module.

The programmer MUST ensure that the declarations are compatible between modules. In other words, all "extern" declarations must match the external declaration (declaration without "extern") by having the same type, size, and amount of indirection. Otherwise, LC may access the variable in incorrect ways.

The extern statement may also be used to declare what a function returns before it is defined in the program. This "forward" declaration allows a function which returns something other than a signed integer to be defined after it is used. If the forward declaration is not given and a function is as-yet-undefined, the compiler assumes that the function returns a signed integer.

STORAGE CLASS - STATIC

Static objects are stored in declared, fixed memory space. Their behavior is the same as that of external variables; their scope is more limited, however. Static variables declared outside of a function can only be accessed by functions within the module being compiled. Other (separately compiled) modules cannot get to them by declaring them "extern". Static variables declared outside of all functions are accessible to all functions within the module. Static variables declared within a function are similar in scope to auto and register variables. They can only be accessed in the block in which they are declared. Thus, two static variables with the same name may be declared in different functions.

Functions may also be defined as "static", making them only accessible from within the current module. However, since LC is a one-pass compiler, the definition of a static function must precede any reference to the static function. This is because the compiler assumes that an as-yet-undefined function is an external function.

STORAGE CLASS - DEFAULTS

When a variable is declared by only stating the storage class:

```
auto x1;  register x2;
extern x3;  static x3;
```

the variable type is assumed to be "int". This is a perfectly acceptable shorthand way to make integer declarations.

When the declaration of a local (declared within a function) variable has no storage class, LC assumes that the variable is an auto variable. A function declared within another function body is assumed to have a storage class of external. The compiler regards the declaration as if an "extern" statement preceded it.

External declarations which do not have a storage class declared are special entities. They belong to the implicit class, "external", and may be referenced from other (separately compiled) modules which declare the variable as "extern".

EXPRESSIONS

One of the most powerful features of the C language is its expression capabilities. The amount of work that can be done by one expression is sometimes mind-boggling. A quick example:

```
(end_of_file = (c=getc(file))==EOF)) ? fclose(file) : ++count ;
```

This convoluted statement will get a character from a file and place it in the variable, "c". The character is compared to the value "eof" which indicates end of file; the result, true or false, is placed in the variable, "end_of_file". Finally, if it was the end of the file, the file is closed. Otherwise, a counter variable, "count", is incremented to provide a count of the characters read.

The example was a bit exaggerated, and expressions this complex can be quite hard to understand. Two statements must be made about the complexity of expressions in the C language.

> The programmer who does not fully know and use C's expression capabilities is seriously handicapped, unable to use the full power of the C language.

On the other hand, a quotation from THE ELEMENTS OF PROGRAMMING STYLE by Kernighan and Plaugher is appropriate:

> "Everyone knows that debugging is twice as hard as writing a program in the first place. So if you're as clever as you can be when you write it, how will you ever debug it?"

The word "maintain" could be substituted for "debug" in the quote above, and it would still be valid. You must be able to understand later what you wrote into your program. If others are going to have to maintain your program, the principle of KISS (Keep It Simple, Stupid) should prevail. This is not intended to discourage the use of complex expressions. Just keep in mind that the more operators involved in an expression, the more difficult it is to properly place parentheses and keep the precedence of operators straight.

There are two kinds of expressions in many computer languages: logical expressions and arithmetic expressions. Logical expressions are usually for comparing things and for making choices. The result of a logical expression is either true or false. Arithmetic expressions result in a number. Usually an assignment to a variable is made to save the result of the arithmetic expression, or it is passed as an argument. In many language implementations, only one type of expression may be used in certain contexts. For instance, the BASIC program statement:

```
1000  A = ( C <= B )
```

attempts to assign to A the result of the comparison C to B. This is not

allowed in many implementations because they are expecting an arithmetic assignment. Even if some BASIC's allow it, it is best not to do this type of assignment, in order to keep programs relatively portable.

Another situation is shown in PASCAL:

```
IF A := (B < C) THEN BEGIN
```

where the PASCAL compiler expects a boolean expression between IF and THEN. Even if A is a boolean variable this assignment is not allowed in most PASCAL compilers. This is not intended to denegrate PASCAL. There are good reasons why the authors of PASCAL did things this way. However, the C language does not draw distinctions between types of expressions within the context of the program. The distinctions are made in the types of operators instead.

PRIMARY EXPRESSIONS

The elements which are manipulated by operators in an expression are called primary expressions. The basic elements which make up a primary expression are identifiers, constants, and strings. Identifiers are the names of variables and functions. Function and array identifiers effectively resolve to the address of the function or array, while all other variable identifiers resolve to the contents of the variable. Constants are character or numeric (decimal, hex, octal) values. Strings resolve to the address of the first character of the string.

The operators which LC provides for stating primary expressions group left to right. This means that the left-most operator is interpreted first. The three primary operators supplied by LC are: isolating parentheses, subscripting, and function invocation.

```
(expression) /* isolating parentheses */
primary_expression [expression] /* subscripting */
primary_expression (expression_list) /* function invocation */
```

ISOLATING PARENTHESES

When the order in which an expression is to be evaluated conflicts with the precedence of operators, the isolating parentheses provide a way around the conflict. The expression within parentheses is evaluated first, before the result of the enclosed expression is used in any expression outside the parentheses. For example, when predicting the percentage of up-time for any equipment, the following formula is used:

$$\text{availability} = \frac{\text{MTBF}}{\text{MTBF} + \text{MTTR}}$$

MTBF = mean time between failures
MTTR = mean time to repair

When writing this formula into a C expression a conflict occurs because the

division operator takes precedence over the addition operator. If the expression is written like this:

```
up_time = mtbf / mtbf + mttr
```

the result will always be mttr plus one. This is because the division is done before the addition. To avoid this, the expression can be stated as follows:

```
up_time = mtbf / (mtbf + mttr)
```

to achieve the correct result.

Parentheses can be used on either side of an assignment operator. At the risk of confusing the reader with as-yet undefined operators, we nevertheless provide an example using pointers. In certain cases during the use of pointer arrays, indirection must be performed before subscripting into the data item. Since subscripting takes precedence over indirection, this kind of expression must be written as follows:

```
example(arg)
     char *arg[]; /* pointer to a char pointer array */
     {
          /* wrong way - accesses third pointer */
          /*   instead of third character. */

          *arg[3] = Ø ;

          /* right way - zero's the third character of */
          /*   first string   */

          (*arg)[3] = Ø ;
     }
```

SUBSCRIPTING

Subscripting is denoted by a subscript in brackets following a primary expression:

```
primary_expression [subscript]
```

If the primary expression is an array name, or a pointer to an array, the subscripted expression returns the element denoted by the value of the subscript. C arrays are subscripted from zero, i.e, the first element in an array is numbered zero.

Function identifiers may not be subscripted. A primary expression denoting an array of pointers to functions may be subscripted. The primary expression must indicate the size of the object being subscripted (char, int, pointer) or the subscript will produce an error message. For example:

```
x = 25[3];
```

is invalid.

FUNCTION INVOCATION

A primary expression followed by parentheses will cause the function denoted by the primary expression to be called. Arguments may be passed to the invoked function by placing them in the parentheses, separated by comma's. LC is very liberal about the primary expression. ANY valid primary expression can be invoked as a function, regardless of the type of the primary expression. Thus, it is perfectly acceptable to write:

```
(Ø xØØ6Ø)();
```

to call the function at hex location ØØ6Ø (This would call the @PAUSE routine in the TRS-8Ø ROM).

Any number of arguments can be passed to the called function. Care must be taken that the number of arguments passed is the number that the function expects. Otherwise unpredictable behavior may result (certainly not correct behavior). If a variable number of parameters must be passed, then a control indicator must be passed to tell the called function how many arguments there are (for example, the fprintf() and printf() functions in the standard library). All arguments listed in a function invocation must appear on the same line in the LC source file. This is not a limitation imposed by the C language, but by the LC implementation. Arguments can be any valid LC expression, including other function calls. The arguments are evaluated from right to left, i.e., the right-most expression is evaluated first. The programmer should not rely on this order of evaluation since some other implementations of the C language evaluate them left to right. Statements like this one:

```
funk( arg++, arg2[arg] );
```

will cause different elements of arg2 to be passed to funk() when different C compilers are used. Stay away from this sort of trickery if you can.

UNSUPPORTED PRIMARY EXPRESSIONS

The primary operators, "." and "->", which are used with structures and unions are not supported (structures and unions are not supported either).

UNARY OPERATORS

Unary operators operate on one object (hence the name). If more than one unary operator operates on the same object, the operators are evaluated right to left. The unary operators supplied by LC are:

OPERATOR	OBJECT	DESCRIPTION
*	expression	indirection, means "object at..."
&	lvalue	pointer, means "address of..."
-	expression	negates the expression, "minus expression"
!	expression	logical complement, "not expression"
~	expression	one's complement of expression
++	lvalue	increment and save in lvalue
--	lvalue	decrement and save in lvalue

All unary operators must appear before (prefix) the object, except the increment and decrement operators. The "++" and "--" may appear after (postfix) the object also. The term "lvalue" means an expression which evaluates to the address of a data element or pointer field. Constants, function identifiers, and array names are not lvalues. The term derives from the observation that "lvalues" are the only expressions allowed on the left side of an assignment expression.

'*'

The indirection operator can only operate on a pointer expression. Its meaning is effectively "object at ..." The address contained in the pointer is the address of the object referred to by this type of expression. For example,

```
see_pointer (pointer)
     char *pointer;  /* a character pointer */
     {
          /* first print the address passed in pointer */
          printf("address is: %d  ",pointer);
          /* now print the data at that address */
          printf("data is: %d  ", *pointer);
     }
```

will print both the address (contents of the pointer variable) and the data at that address (result of the indirect expression).

'&'

This unary operator effectively means "address of..." or "pointer to...". It evaluates to the address of the lvalue it precedes.

'-'

When the unary negation operator precedes an expression, the result is the two's complement negative of the value of the expression. When the '-' precedes an unsigned or pointer expression, the one's complement of the value is taken. Since all expressions in LC are evaluated using 16-bit arithmetic, the expression is negated for the full 16 bits.

'!'

The unary logical complement operator, or "not" operator evaluates to false if the expression is true and to true if the expression is false. False is defined as Ø and any non-zero value is considered to be true. However, all LC operators which result in true or false use one (1) as the value for true. Thus, the least significant bit of the result indicates true or false.

'~'

The one's complement operator inverts every bit in the expression. No regard is given to the type of the expression.

'++', '--'

The increment and decrement operators may be used either before (prefix) the operand or after (postfix) the operand. The operand must be an lvalue or lvalue expression. In either case the contents of the lvalue is incremented or decremented and stored back into the lvalue. The difference between prefix and postfix is whether the result of the expression is incremented or not. Prefix means that the value after the increment or decrement is the result of the expression. Postfix means that the value returned by the expression is the value before the increment or decrement.

UNSUPPORTED UNARY OPERATORS

The "(type_name)" and "sizeof" operators are not implemented in LC.

BINARY OPERATORS

Binary operators act upon two expressions together. The type of the result depends on the type of the two expressions. If the type of any of the expressions is "char", "short", or "long", it is treated as an integer. If one expression is unsigned, the other expression is treated as unsigned as well, and the result is unsigned. If one expression only is a pointer, the result of the expression is a pointer of the same type. If both expressions are pointers, the result is unsigned.

When several binary expressions are concatenated together (without isolating parentheses) the order in which the binary expressions are evaluated depends on the precedence of the operators in the expression. In the expression,

```
a + b * c
```

the evaluation of "b * c" precedes the evaluation of the addition, since multiplication has a higher precedence than addition. The expression is evaluated like this:

```
a + (b * c)
```

As previously described, isolating parentheses can be used to change the order of evaluation. To have the addition performed first, the expression can be written:

```
(a + b) * c
```

Each class of operands is described below in order from the highest precedence to the lowest. Note that when all the operators in a complex expression have the same level of precedence they are evaluated in a certain order; right to left or left to right. It can be said that a class of operators "group" left to right, or right to left. If the order of evaluation between like operators does not matter, the operator is said to be associative. Here is an example of how the order of evaluation affects an expression:

```
a / b / c / d
```

This expression is evaluated a follows:

```
(((a / b) / c) / d)
```

Thus, the division operator is said to group "left to right".

PRECEDENCE OF BINARY OPERATORS
(Highest to lowest)

MULTIPLICATIVE OPERATORS	- group left to right
expression * expression	multiplication
expression / expression	division
expression % expression	modulus (remainder)
ADDITIVE OPERATORS	- group left to right
expression + expression	addition
expression - expression	subtraction
SHIFT OPERATORS	- group left to right
expression << expression	shift left
expression >> expression	shift right
RELATIONAL OPERATORS	- group left to right
expression < expression	less than
expression > expression	greater than
expression <= expression	less than or equal to
expression >= expression	greater than or equal to
EQUALITY OPERATORS	- group left to right
expression == expression	equal to
expression != expression	not equal to
BITWISE AND OPERATOR	- associative
expression & expression	bitwise and
BITWISE EXCLUSIVE OR OPERATOR	- associative
expression ^ expression	bitwise exclusive or
BITWISE INCLUSIVE OR OPERATOR	- associative
expression | expression	bitwise inclusive or
LOGICAL AND OPERATOR	- groups left to right
expression && expression	logical and
LOGICAL OR OPERATOR	- groups left to right
expression || expression	logical or
CONDITIONAL OPERATOR	- groups right to left
expression ? expression : expression	
ASSIGNMENT OPERATORS	- group right to left
lvalue = expression	simple assignment
lvalue <op>= expression	compound assignment
(<op> is any binary operator except logical, relational, or conditional operators)	

'*', '/', '%'

The multiplicative operators take precedence over all other binary operators and group left to right. When the result of multiplication overflows 16 bits, the left-most (high-order) bits are truncated. Since integer division is used, the fractional portion of the result is lost. The result of division is always truncated toward zero. The modulus operator returns the value of the remainder in the integer division of the two expressions.

'+', '-'

The additive operators result in the addition or subtraction of the two expressions. In subtraction, unsigned subtraction only takes place when both expressions are unsigned. If one of the expressions is a pointer and the other is not, the other value is adjusted to reflect the size of the object pointed to. Thus, if "p" is a pointer, "p + 3" returns the address of the fourth object pointed to by "p". If p points to integers, then LC automatically doubles the offset to account for the two-byte elements.

'<<', '>>'

The shift operators shift the left-hand expression by the number of bits indicated in the right-hand expression. Zeroes are shifted in to replace the bits shifted out. If the right-hand expression is negative or zero, no shifting takes place. If the right-hand expression is 16 or more, the result is always zero.

'<', '>', '<=', '>='

Relational operators result in a true (1) or false (Ø) value, depending on the indicated condition.

'==', '!='

The equality operators, "equal" and "not equal", respectively also return true (1) or false (Ø) depending on the two expressions' equality.

'&'

The bitwise AND operator does a bitwise AND with the two expressions. Each bit position in the result will be set to be one if and only if both corresponding bits in the expressions are equal to one. This is useful for isolating individual bits within a word by using a "mask" as one of the expressions. Any bit in the mask which is set to zero will cause that bit in the result to be zero. Any bit set to one will cause the bit in the other expression to remain the same.

'^'

The bitwise exclusive OR operator. Each bit in the result of an exclusive OR is set only if the corresponding bits in the expressions are opposite, i.e., 1 and Ø, or Ø and 1. If they are the same, that bit in the result will be zero. This can be used to complement bits, using a "mask" expression. Any bit which is 1 in the mask will cause the corresponding bit of the other expression to be complemented in the result. Any mask bit which is Ø will pass the corresponding bit unchanged into the result.

'|'

The bitwise inclusive OR operator. Each bit in the result will be set to 1 if either of the corresponding bits in the expressions are equal to 1. This can be used to set any particular bit in an expression to one by using a "mask" expression. If a bit in the mask is equal to 1, then the corresponding bit in the result will be set to 1. If a bit in the mask is equal to Ø, then that bit in the result will be the same as in the expression being manipulated.

'&&'

The logical AND operator results in a true (one) or false (zero) condition, depending on the relationship of the two expressions. The result is true only if both expressions are true (non-zero). Moreover, if the first expression is false, the second is never evaluated.

'||'

The logical OR operator returns a true (1) result if either of the expressions is true (non-zero). If the first expression is true, the second expression is not evaluated.

'? :'

The conditional operator gives the C expression repertoire the equivalent of an if-then-else construct. It can technically be classified as a binary operator since only one of the last two expressions is evaluated. The first expression is evaluated as true (non-zero) or false (zero). Then, if the first expression was true (non-zero), the second expression is evaluated as the result of the expression. Otherwise, if the first expression was false (zero), the third expression is evaluated for the result. The conditional operator groups right to left:

```
a ? b : c ? d ? e : f : g
```

is evaluated in the following manner:

```
a ? b : (c ? (d ? e : f) : g)
```

Either or both of the second and third expressions can contain conditional expressions.

'=', '+=', '-=', '*=', '/=', '%=', '<<=', '>>=', '&=', '^=', '|='

The assignment operators place the result of the right-hand expression into the object denoted by the left-hand expression, after performing the indicated operation with the contents of the lvalue when an assignment operator other than '=' is used. The simple assignment, '=', places the result of the right-hand expression unchanged into the object denoted by the left-hand expression. The compound assignment operators have the form:

expression_1 <op>= expression_2

and is evaluated like this expression:

expression_1 = expression_1 <op> expression_2

The first form is more efficient since expression_1 only need be evaluated once.

LC STATEMENTS

LC statements are used to specify the action to be taken by the program. The statements given in the program are executed one after the other. Certain statements (conditional and looping statements) will direct the order of and conditions for execution of other statements. Some definitions of statements in the following text require that a substatement be included in the statement. Any place where a substatement is required there may be one simple statement or more than one statement combined in a compound statement.

SIMPLE STATEMENTS

Simple statements are of three types: expression, declarative, and control. The declarative statements are described fully in the previous sections on functions and variables. The type, size and scope of functions and variables are declared in declarative statements.

A simple statement always ends with a semicolon. The semicolon is the STATEMENT TERMINATOR. It is not a statement separator as in the PASCAL language. It is always required at the end of a simple statement.

COMPOUND STATEMENTS

The left and right brace characters, "{"and "}", are used to indicate the beginning and end (respectively) of a compound statement. A compound statement, also called a block, can be used anywhere that a simple statement may be used. Thus, wherever LC's syntax requires a statement, more than one statement may be given by enclosing them in braces. Within the compound statement there may be any combination of simple and compound statements.

The compound statement has the format:

```
{   <declarations>  <statements>     }
```

No declarations or statements are required, although the use for an empty block would be as a null statement of sorts. The declarations should appear before any statements. Any of the statements may in turn be another compound statement. No semicolon is required after the compound statement.

The only place where a compound statement is required instead of a simple statement is in the body of the switch-case statement. The body of a function is one compound statement. Here are some examples of compound statements:

```
func()
{    /* the body compound statement */
     a=b;       /* simple statement */
     if (a>c)
     {    /* another compound statement */
          c=a;
          b=a;
```

```
        }    /* end of compound statement */
        return a+b+c;
    }    /* end of function body compound statement */
```

NULL STATEMENT

A null statement is a sort of place-holder. LC requires that a statement be given in certain places. If no action is needed in the place required then the null statement can be used. No action is taken by the null statement.

The null statement is simply a statement terminator (semicolon) by itself, with no preceding statement.

```
null()
{    /* do-nothing function */
     ;    /* a null statement */
}
```

EXPRESSION STATEMENT

An LC expression followed by a semicolon is called an expression statement. The expression is performed when it is encountered. LC will allow an expression that has no assignment in an expression statement, even if it does nothing. Expression statements are used to assign values to or modify values of variables, or to invoke functions. Some sample uses of expression statements:

```
retcode = call_function() ;   /* call a function */
a = b = c = Ø ;     /* make a, b, c equal to Ø */
++counter ;    /* increment a counter */
```

IF STATEMENT
---

if (expression) statement

if (expression) statement
else statement

The "if" statement gives the programmer the capability to decide whether a statement will be executed. The criterion for the decision is the result of evaluating the expression. The expression may be any valid LC expression. If the expression evaluates to true (non-zero), then the statement is executed. If the expression evaluates to false (zero), then the statement following the "else" (if any) is executed.

"If" statements may be nested, i.e., the statement within an "if" statement may be another "if" statement. Too much nesting of "if" statements can be hard to follow, so moderation is advised.

Some examples of "if" statements:

```
if (x < 0)  x = -x ;      /* absolute value of x */

if (i<=0) {  i=x;   b=a;  }   /* compound statement */
else --i;       /* and an else clause */

/* nested if statements */
if (past_twelve)
     {if (before_six) say("good afternoon");
     else say("good evening");}
else say("good morning");
```

SWITCH-CASE STATEMENT
---

switch (expression) { <switch_statement> ... }

switch_statement = statement
case constant_expression :
default :

The switch-case statement allows program execution within the <switch_statement> to be determined by the case and default prefixes. The expression in the switch statement is evaluated first, then, if any of the constant_expressions match the result, execution begins immediately past that case prefix. If none of the cases match the result and there is a default prefix, then execution begins at the default prefix. Otherwise, when no matching case is found, no statements in the switch_statement are executed.

The switch-case statement MUST have a compound statement as its substatement. This is the only case where this is true. The default and case statements may occur in any order within the body of the switch-case. However, A CASE OR DEFAULT MUST PRECEDE THE FIRST STATEMENT in the switch-case. If this is not done, none of the cases will ever be executed (LC limitation).

The break statement is used to exit the switch_statement.

```
switch (month)
{    case january:  case october:  case december:  case july:
     case august:   case march:    case may:
          days = 31; break;
     case september:  case april:  case june:  case november:
          days = 30; break;
     case february:
     {    if (leap_year) days = 29;
          else days = 28;
          break;
     }
     default: days = 0; error = true;
}
```

WHILE STATEMENT

while (expression) statement;

The most basic form of looping is provided in LC by the "while" statement. Simply stated, while the expression results in a true (non-zero) value, "statement" (also called substatement) is executed. The expression is evaluated before each time the substatement is executed. Therefore, the substatement may be executed from zero to any number of times depending on the expression.

If more than one simple statement must be placed in the loop substatement, then the substatement must be a compound statement. The break statement can be used to exit the loop from within the statement. The continue statement can be used to continue directly on to evaluate the expression, skipping the rest of the substatement, from anywhere within the statement.

```
while (driving)   watch(the_road);

while (jogging)
{    take(a_step);
     breathe();
     if (too_tired)  break;
}
```

DO STATEMENT

do statement while (expression) ;

"Do" differs in only one way from the while statement - the expression is evaluated after the statement is executed. Therefore the substatement will always be executed at least once. The substatement will be repeatedly executed until the expression evaluates to false (non-zero).

```
do  anything();
while (there_is_still_time) ;

/*  shuffle routine  */
do
{    cut_the_cards();
     shuffle();
}
while ( ! ready_to_deal ) ;
```

FOR STATEMENT

for (expr_1 ; expr_2 ; expr_3) statement

The for statement is a looping statement which provides a convenient place for initializing, testing, and incrementing loop control variables. The format shown above can be rewritten using the while statement:

```
expr_1 ;
while ( expr_2 )
{    statement
     expr_3 ;
}
```

Expr_1 is evaluated once before the loop is entered. The test expression, expr_2, is evaluated before each execution of the substatement. If it results in a false (zero) value, the loop is not executed and execution continues to the next statement. Expr_3 is evaluated after each time the substatement is executed.

Both expr_1 and expr_3 can be more than one expression, separated by commas. Expr_2 can only be one expression and should result in a logical value.

Due to limitations of the LC implementation, all three expressions in parentheses must be on the same line.

```
for ( c = 'A' ; c <= 'Z' ; ++c )
     putchar(c);    /* print the letter */
/*  "Now I've said my ABC's ..."  */
```

BREAK STATEMENT

break ;

Break is used to exit any "while", "do", or "for" loop and to exit the body of a "switch" statement. Whenever a break statement is encountered, execution immediately goes to the next statement past the loop or switch statement. Break is illegal outside of any loop or switch compound substatement. For an example of the use of break in a switch statement, see the section on switch-case above.

```
strscan(c,s)
     char c,*s;
{    /*  find character c in string s  */
     while (*s != c)
     {    if (*s == Ø)  break ;  /* end of string */
          ++s ;  /* next character */
     }
     return s ;
}
```

CONTINUE STATEMENT

continue ;

The continue statement is used to skip the remaining statements in a compound loop substatement. In a "while" or "do" statement, execution continues at the test expression. In a "for" statement, execution continues at the reinitializing expression (the third expression). The continue statement is illegal outside of any loop statement.

```
/*  convert to lower case  */
while ((c = getchar()) != eof)
{    if ( c < 'A' || c > 'Z' )  /* not an uppercase character */
     continue ; /* doesn't apply */
     c = tolower(c) ;
     putchar(c) ;
}
```

RETURN STATEMENT

return ;
return expression ;

The return statement causes the currently executing function to end. If an expression is provided, then the result of the expression is returned as the value of the function. The returned value is undefined if no expression is provided in the return statement. The return statement is not required to return from a function. When no statements are left (the bottom of the function body is reached), the function automatically returns as if a return statement with no expression were encountered. Return statements are needed when a value must be returned or when the return must take place before the end of the function.

```
square (num)
/*  square a number  */
{    return  num * num ;    }

getline(buf)
     char buf[] ;  /* line input buffer */
{
     /* check for a valid file pointer */
     if (file_pointer == NULL)
     {    buf[Ø] = '\Ø' ;  /* put a null string in buf */
          return ;  /* back to caller */
     }
     fgets(buf, bufsize, file_pointer);
}
```

GOTO STATEMENT

goto label;

The goto statement causes an unconditional branch to the statement identified by label. The labeled statement must be contained in the current function. It is illegal to attempt a goto to a statement in some other function. An attempt to do so will result in an error during the assembly phase. The following example illustrates the use of goto (note: it is strongly recommended that you avoid the use of the goto statement):

```
rest(time)
int time;
{
     if ( time > 23ØØ ) goto sleep;
     else return;

sleep: for ( ; ; )
     ;
}
```

LABELED STATEMENT

label: statement;

Any statement can be prefixed with a label. This construction is usually used to target the argument of a "goto" statement. The format of a label is a valid identifier followed by a colon. The following are labeled statements:

```
calculate: i += 1Ø;

bigblock:  i = j = k = l = m = n = o = p = Ø;
```

LC OPERATION

LC takes C source code as input and generates an EDAS Version IV compatible assembler source file as output. Thus the output of the compilation process must be assembled and linked with any required run-time library module before it can be executed. The assembly and linking process is performed using the EDAS IV assembler. A Job Control Language (JCL) file, "LC/JCL", is provided to present the compilation and subsequent assembly as a job stream to the operating system. The JCL procedure requires minimal entry of commands by the programmer to create an executable CMD file. The JCL file is:

```
. Batch creation of a runnable ELSIE program.
. Format is:   do lc (file=<progname>,{show})
//if show
lc #file# +l
//else
lc #file#
//end
edas (jcl,abort)
llc
c/cprogram/#file#
//if show
a#file# -we
//else
a#file# -nl
//end
b
. completed compilation
```

If you want to compile a program called "myprog" and generate the finished CMD file with only one statement, then the command:

```
DO LC (FILE=MYPROG)     /* See ATTENTION on page 3-11 */
```

lets you sit back and relax while the machine does all of the work. If the /ASM and /CMD files do not already exist, you may want to enter a drive specification as follows:

```
DO LC (FILE=MYPROG:1)
```

which looks for input and places output on drive one (1).

The first stage of the LC language process is, of course, to create an LC source file. The editor that is a part of EDAS IV is provided for this purpose. In order to use EDAS for the creation and maintenance of LC source files, execute EDAS with the command:

```
EDAS (LC)
```

The EDAS manual should be consulted for all operations concerning the editor or assembler functions.

The second stage of the LC process is the compilation of the LC source using the LC/CMD compiler. Use the JCL procedure shown above. However, if you want to take direct control of the operation, you can execute LC directly. LC is executed when a command line beginning with LC is typed at the "DOS ready" prompt or encountered in a JCL file. The format of the command line is free-form - simply a list of input file specifications and option switches. The command syntax is as follows:

```
LC filespec {filespec...} {switch} {switch...}

filespec  -  A file specification for the input file(s).
             A maximum of 8 filespecs may be passed.

switch    -  Represents an optional compiler switch(es)
             These switches are preceded with either a
             plus sign (+ = on) or a minus sign ( - = off).
```

The compiler is executed by entering a command line such as:

```
LC CPROGRAM:2 +LIST
```

which compiles the LC source file, "CPROGRAM/CCC", and generates the output file, "CPROGRAM/ASM", on drive 2. The "+LIST" switch specifies that you want the LC source code listed to the screen during the compilation process.

The switches allow the user to control certain features of the compiler. Switches and filenames may be intermixed in any order on the command line. The simplest compilation command would simply be "LC PROGNAME" which compiles the file, "PROGNAME/CCC", and generates the output file, "PROGNAME/ASM".

FILE SPECIFICATIONS

There may be up to eight input file specifications given on the command line. They are processed by the compiler in the order they occur (left to right). If no extension is given for a source file, the default extension "/CCC" is assumed. It is recommended that you establish your LC source files with this file extension for uniformity and standardization. If you use "EDAS (LC)" for your LC source code maintenance, the use of the /CCC extension is automatic.

The output file specification defaults to the same name as the first input file specified. LC will append the file extension "/ASM" to this name. The drive specifier, if any, of the first input filespec is used as the drive specifier of the output file. The drive specifier should be given if the output file must be written to the same drive as the input file. The LC OUTPUT option may be used to specify a different file name or change the destination drive number. Assembler source code output may be suppressed by turning off the OUTPUT option. This can be helpful for quickly checking syntax without generating an output file.

COMPILER SWITCH OPTIONS

Compiler option switches are turned on or off by a '+' or '-', respectively, followed by the name of the switch. For example, "+list" causes the LC source code to be presented to the standard output device during the compilation. The compiler regards any command line argument not beginning with a plus or minus as an input file specification. Only the first letter of the switch is examined, so partial spelling (or misspelling) is accepted. Certain switches have operands which are specified by following the option name with '=' and the operand. For instance, "+output=myfile:3" will cause the output by the compiler to be written to "MYFILE/ASM" on drive 3 instead of the filename that would have been the default.

COMMENT

This switch controls whether the original LC source code will be written to the assembler output file as comments. The normal default is ON. The C-source appearing as comments may be instrumental in your understanding the compiler output as it generates a minimally commented assembly source program.

GLOBAL

GLOBAL controls the definition of external variables. If the switch is turned off, variables declared external are not defined in the assembler output. If turned on, external variables are defined in the assembly module output. This switch defaults to on. For more information on using the GLOBAL option, see the ADVANCED TOPICS chapter.

LIST

The standard output would normally receive minimal feedback during the compilation process. If the LIST option is turned off, LC will write each function name followed by a period representing each statement in the function. An example of this is:

```
main() {.............}.
move() {..........}.
min() {....}.
max() {....}.
p() {.........................}.
No errors found
```

If you want to see the source code as LC is compiling, you must specify the "+l" option, since the default is "-LIST". Note that since listing output is to "standard output", it is subject to I/O redirection. Thus, if you want a printer listing, for whatever reason, you could specify ">*PR" in the LC command line.

OUTPUT=SPEC

This switch controls the output of the compiler to the output file. If the switch is off, no output file is generated. However, if it is on, but no SPEC is given, LC appends "/ASM" to the name of the first input file in order to create the output file specification. When a file specification is given for SPEC, it becomes the name given to the output file. A default extension of "/ASM" is inserted if no extension is given. If only a drive specification (":D") is given, the output file is written to that drive, with the same file name as the first input file. This switch defaults to on with no SPEC.

PAUSE

When this switch is on the compiler will stop when any errors are found and displayed. Any key except BREAK will continue compilation. BREAK will abort the execution of the compiler at any time if this switch is on. If LC was invoked from JCL, the JCL will also be aborted. This switch defaults to on.

CREATING A CMD FILE

Once the LC compiler has compiled your program into assembly language, you need to use the EDAS assembler to create the CMD file. In order to provide the proper initialization in the CMD file and ensure that all necessary runtime routines are linked with your program, a special assembler file, LC/ASM, has been provided. A listing of the file follows to aid in illustrating its functions:

```
;LC/ASM - Ø9/Ø9/82
;*=*=*
;       This module is assembled to create the
;       run-time /CMD program file. The line:
;                *GET CPROGRAM
;       fetches the file containing your "main"
;       program compiled by LC. Separately
;       compiled C-source modules can be fetched
;       by adding additional *GET statements.
;       If you have created a user library(ies),
;       add additional *SEARCH statements.
;
        COM '<LC is copyrighted (c) 1982 by Jim Frimmel>
;*=*=*
        ORG     52ØØH
@START: LD      HL,(4Ø49H)          ;P/u Model I HIGH$
        CALL    @MOD13              ;Test for Model I/III
        JR      NZ,$+5              ;Go if Model I
        LD      HL,(4411H)          ;  else use III's
        LD      SP,HL               ;Set stack area
        CALL    @GO                 ;Initialize
        CALL    MAIN                ;Execute user prog
        LD      HL,Ø                ;Set return code
        PUSH    HL
        CALL    EXIT                ;Back to DOS
$$STEMP DEFL    Ø                   ;Init relative storage
*GET LCMACS                         ;Get macros used by LC
*GET CPROGRAM                       ;Fetch user program
        IF      @_INLIB             ;True if #option inlib
*SEARCH IN/LIB                      ;Installation lib?
        ENDIF
        IF      @_FPLIB             ;True if #option fplib
@_FPLIB DEFL    @_FPLIB.OR.FPINIT   ;Force GET of fpinit()
*SEARCH FP/LIB
        ENDIF
*SEARCH LC/LIB                      ;Standard lib always!
$$STORG EQU     $                   ;Set to 1st storage byte
$PROGEND DEFL   $$STORG+$$STEMP     ;Establish program end
        END     @START
```

The LC/ASM file provides a front end that makes your program usable on either a Model I or Model III. This is the file that is assembled. Notice that the bulk of the resulting CMD program is assembled from files via the *GET statement and *SEARCH statements. If you have not yet read chapter five of

the EDAS manual, you may want to temporarily turn to the sections on *GET and *SEARCH located within the fifth chapter.

If your compiled program file was named MYPROG/ASM, it is linked into LC/ASM by changing the statement "*GET CPROGRAM" to "*GET MYPROG". This is done simply with the EDAS editor or is automatic when using the LC/JCL Job Control Language "hands-off" procedure.

For a great deal of your programs, the only LC runtime routines needed will be located in the LC/LIB library. Since all LC programs need some of the routines in LC/LIB, that library is ALWAYS searched during the assembly process. The floating point library, FP/LIB, will be automatically searched if your program requested the floating point library search via an "#option fplib" compiler macro. This is explained in more detail in the LC LIBRARIES chapter.

Many useful routines are stored in the installation library, IN/LIB. This library is not normally searched in order to save you assembly time when you need not refer to the IN/LIB routines. However, it is very easy to force an automatic search of the installation library. All you need to do is specify an "#option inlib" compiler macro in your LC source program (similar to "#option fplib"). For example,

```
#include stdio/csh
#option inlib  /* This statement invokes *SEARCH IN/LIB */
main()
int dot;
{ for ( dot=Ø, dot < 128, dot++ )
     set( dot, Ø );
  exit(Ø);
}
```

will schedule the compilation and assembly of your program with a forced search of the installation library (to resolve linkage to the "set()" function.

COMPILER DIRECTIVES

The LC compiler supports a handful of directives that control various aspects of the compiler during the compilation process. One of these directives, "#include filespec", you will quickly become familiar with. Others may be used less frequently. They, nevertheless, provide additional power in the use of the LC language "system". These directives are:

#include

#include <filespec>

This directive tells LC to insert the file designated by "filespec" into the source stream being compiled. The <filespec> will default to an extension of /CCC if no extension is given. The #include is used quite frequently to merge the STDIO/CSH standard header file into your compilation. An illustration of its use is:

```
/* sample program to illustrate #include */
#include stdio/csh
main()
{
     int x;
     x = -3 + 4 * 5 - 6;  printf("%d\n",x)
}
```

#define

#define <macname> <definition>

The "define" directive is a macro definition. It creates a macro, called "macname", which is defined to be the string of characters following the macname (the definition). The compiler will substitute the string wherever "macname" is found in the LC source stream. It is strongly recommended that macro "macnames" be defined in upper-case characters so that it becomes distinct when looking at your source code.

<macname> must be a valid LC identifier, whereas the <definition> is anything and everything up to a comment or the end of the line. The <definition> is substituted whenever the <macname> is encountered in the LC source code input past the #define of <macname>.

An example of the use of #define follows:

```
#include stdio/csh
#define PRINTX printf("%d\n",x)
main()
{    int x;
     x = -3 + 4 * 5 - 6; PRINTX;
     x = 3 + 4 % 5 - 6; PRINTX;
}
```

#option

#option <optname> {value}

The #option directive is used to pass symbol definitions from the LC source code to the assembly phase. The <optname> must be a valid LC identifier. Value must be a numeric or character constant. Escape sequences may be used in the constant. The compiler translates the #option directive to the form:

```
@_NAME    DEFL    value
```

The "value" is optional (as shown above by apearing within braces). If the value is omitted, the DEFL statement will default to a value of negative one (-1). This indicates TRUE to the EDAS assembler.

The #option directive is used in LC to invoke a search of the installation library, IN/LIB, or the floating point library, FP/LIB. If your application will be using functions in either library, you will need to add the statement(s):

```
#option FPLIB
#option INLIB
```

for the floating point and installation libraries respectively.

LC has reserved additional option names for use with the #option directive. These are:

ARGS - specify that your program {will}/{will NOT} be using command line arguments (argc, argv). LC will suppress the run-time code normally used to process arguments thus reducing the size of your CMD program.

ERRORMSG - specify supppression of I/O errors.

FIXBUFS - specify pre-allocation of buffers for standard I/O.

GETNL - specify strip of newline character in fgets().

KBECHO - specify echo of keyboard input to the display.

MAXFILES - specify the maximum number of concurrently opened files permitted.

REDIRECT - specify that your program {will}/{will NOT} be using standard I/O redirection (>, <, #). LC will suppress the run-time code used to process I/O redirection thus reducing the size of your program.

ZVAR - specify initialization of variables to zero.

For additional information on the use of these options and "#option" in general, read the section on options in the ADVANCED TOPICS chapter.

#asm - #endasm

```
#asm
  .
  transparent assembly language code
  .
#endasm
```

The directive pair, #asm - #endasm, can be used to insert assembly language source code directly within the LC source file. It should be used ONLY when it is ABSOLUTELY necessary to write a routine in assembly language. Remember, any LC source code file that has imbedded assembly language code is generally NOT portable. The more assembly language code you imbed, the less portable your programs become and the more you have to recode when transporting your program to another machine.

All input past the #asm statement is passed unchanged into the output file. Of course, since the output file is an assembly language source file, the statements following the "#asm" should be assembly source statements. The block of assembly statements is ended with the #endasm statement. Please note: the "#endasm" statement must be the first thing on a line, other than tabs and spaces. Otherwise the "#endasm" will not be recognized and the C source code following the "#endasm" will be passed uncompiled to the output file.

This escape to assembly language is provided as a convenient kludge mechanism only. It is not intended to be the normal way of interfacing assembly language functions to a C program. The proper way to interface to assembly language is to place the assembly function in a separate module, perhaps even in a user library if it is to be used frequently. This makes the program easier to transport to other systems, as the machine-dependent code is separated from the program source. See the Advanced Topics chapter for more information on assembly language programming in the Elsie environment.

A SIMPLE EXERCISE

It may prove helpful to you to see a program generated from start to finish. It will be a small one, but nevertheless, one that will exercise all of the steps needed to accomplish the creation of a CMD file. If you have ever entered and/or edited a BASIC program, you are ready to perform this exercise. Before we begin, remember that you will need the LDOS keyboard driver activated. A few of the extra keyboard characters will be needed. It's probably a good idea to refresh your memory as to the key combinations.

character	key combination
[	CLEAR-COMMA
\	CLEAR-SLASH
]	CLEAR-PERIOD
^	CLEAR-SEMICOLON
_	CLEAR-ENTER
{	CLEAR-SHIFT-COMMA
|	CLEAR-SHIFT-SLASH
}	CLEAR-SHIFT-PERIOD
~	CLEAR-SHIFT-SEMICOLON

You must have the book, "The C Programming Language" by Kernighan and Ritchie. Open it to page 15 and note the program shown at the top of the page. You are going to enter it. First, execute EDAS with:

EDAS (LC)

The "LC" parameter tells EDAS to accept lower case input, set tab characters at every four positions, and use /CCC as the default file extension. You will observe the EDAS heading message. Now enter the command, "i". EDAS will go into insert mode and display the first line number. As you enter each line, terminate it with <ENTER>. You should get into the habit of using the <TAB> key (right-arrow) to organize your LC code into neat indentations. It will be shown below as "<T>". Follow the line numbers shown below with the text as found in K & R.

```
 EDAS   YOU TYPE
------  ------------------------------------------------------
ØØ1ØØ   #include stdio/csh <ENTER>
ØØ11Ø   main() /* copy input to output */ <ENTER>
ØØ12Ø   { <ENTER>
ØØ13Ø   <T> int c; <ENTER>
ØØ14Ø   <T> while (( c = getchar()) != EOF) <ENTER>
ØØ15Ø   <T><T> putchar(c); <ENTER>
ØØ16Ø   } <ENTER>
ØØ17Ø   <BREAK>
 >      w clone:d <ENTER>    /* ":d" specifies the drivespec */
New file!
 >      b
```

You have just entered an LC program, saved it under the name "CLONE/CCC:d", and returned to DOS Ready. Now enter the command (with ":d" representing your drivespec):

DO LC (FILE=CLONE:d)

If you have entered the program correctly, LC should compile your program and invoke its assembly to create the CLONE/CMD file. Try out your program according to K & R.

If you want to start understanding the concepts of I/O redirection, turn to page 1-7 of the INTRODUCTION. The program you have just compiled into an executable CMD command is used to illustrate I/O redirection. Try out the examples shown.

```
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
>>>>>                                          <<<<<
>>>>>              ATTENTION                   <<<<<
>>>>>                                          <<<<<
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
```

If you are invoking LC with the **DO LC (FILE=MYPROG)** command and the EDAS assembly aborts with a Job Control Language, "Job aborted" message, you most likely mistyped one or more identifiers (variable names, function names, etc). You can easily discover the error by reinvoking the procedure with the command,

DO LC (FILE=MYPROG,SHOW)

STANDARD LIBRARY

The standard library, LC/LIB, is a collection of useful functions that allows the user to interface with the world external to the program, without having to know the specifics of the particular environment that the program is running in. Thus, a program can be transported in source form to a different computer under a different operating system. Only the standard library need change between systems.

The library is defined in a device independent way so that a program can use any device for input or output. Since any file is defined as a sequence of bytes, all devices can be interfaced to as files. LDOS already provides this type of device independence, thus the implementation of LC I/O is totally compatible with normal LDOS files.

The standard library also provides functions to perform machine dependent operations, such as memory allocation and character set operations.

Every program in executable form under LDOS requires a small run-time module in order to open standard I/O and provide I/O redirection, initialize the I/O and memory allocation functions, and to provide basic operations that LC programs require, such as MULTIPLY, OR, and AND.

The library is constructed as a Partitioned Data Set. LC functions are stored in assembler source form as members of this data set. The standard library is accessed by the EDAS assembler from the "*SEARCH LC/LIB" command. Any member needed by your C language program is automatically linked with your program during the assembly phase of the LC compiled output. See the chapter titled, "OPERATOR'S GUIDE", for more information.

ALLOC()

This function is used to allocate a memory block. Its syntax is:

```
ptr = alloc ( nbytes );

nbytes     -  unsigned number of bytes needed.

ptr        -  address of the block allocated.
```

Alloc() is used to dynamically allocate memory during program execution. The complementary function, free(), is used to release memory allocated through alloc(). Alloc may be used to get table or buffer space when the amount of memory space available is unknown, such as a program designed to run in any size memory machine (16k, 32k, 48k). The programmer can call alloc() with decreasing size of requested space until the space is allocated.

RETURN CODE

If a memory block has been allocated, the value returned is the address of the memory block. If insufficient memory is available to satisfy the allocation, alloc() will return a null (Ø).

WARNINGS

The program must not access memory outside of the area allocated. File access routines use alloc() and free() to establish and release File Control Areas (FCA's). The programmer cannot assume that memory not allocated is free for use, since later file opens may cause memory overlays. It is advised that the programmer always use alloc() and free(), or sbrk() for all dynamic memory accessing.

EXAMPLE

```
symtbsz -= symtbsz % symsiz;/* make integral */
if ((symtab = alloc(symtbsz)) == Ø)
   abend("not enough memory");
glbptr = startglb = symtab;
```

DATA CONVERSIONS: ATOI() ITOA() XTOI() ITOX()

These functions are used to convert character strings of digits (decimal or hexadecimal) to their integer value and vice versa. The syntax is as follows:

```
int = atoi( decs );

itoa( int, decs );

int = xtoi( hexs );

itox( int, hexs );
```

int - is an integer value.

decs - is a string containing decimal digits <Ø-9>

hexs - is a string containing hexadecimal digits <Ø-9>, <A-F>, or <a-f>.

These standard C functions are used to convert integer values to their character string image and the converse. Functions are provided to deal with character strings containing either decimal or hexadecimal digits. Left truncation of the integer value takes place if an excess number of digits is present (i.e. int=xtoi("11ØØØ") would result in the integer value of 4Ø96 decimal). Conversion of a decimal string will be modulo 65536.

Note: the C itoi() function has not been implemented for obvious reasons.

RETURN CODE

There is no return code.

EXIT()

This function is used to exit your LC application and return to DOS. Its syntax is:

```
exit( code );

code       - integer return code.
```

Exit() allows the user to exit cleanly from a program and control the consequences of exiting. Passing a zero (Ø) for the return <code> to exit() indicates normal program termination, causing exit() to take the LDOS normal exit. If a non-zero <code> is passed, exit() will take the error entry into LDOS, thus aborting any JCL processing in effect.

If the terminating program was invoked by the cmd() function, the value passed to exit() will be the value returned from cmd(). An exception is if a negative value value is passed to exit(), in which case a negative one (-1) is returned to cmd().

Exit() closes all open files before returning to LDOS.

RETURN CODE

Exit() does not return to the caller.

WARNINGS

For compatibility with other C language systems, the programmer should not depend on exit() to close the program's files (other than standard I/O files).

FCLOSE()

This function is used to close an open file. Its syntax is:

```
retcod = fclose ( fp );

fp          -  the file pointer.
```

Fclose() is used to close an open file and to free the file control area (FCA) for later use. The <fp> passed to fclose must have been obtained from fopen(). In LC, exit() also closes files; however, the programmer should use fclose() to ensure compatiblity and portability.

There is a limited number of files (determined by the MAXFILES compiler option), including standard files, that may be open at one time. Fclose() is used to free FCAs so an unlimited number of files may be accessed one after the other.

RETURN CODE

The "retcod" will be non-zero if no error was detected in the closing operation. If an error was detected, then "retcod" will be zero (Ø).

WARNINGS

The value passed to fclose() must be a valid file pointer. If it is not, unpredictible things, such as destroyed disk files, reboots, etc., can result.

EXAMPLE

```
if (lastc != Øx1a) putout(Øx1a);
fclose(fp1);  fclose(fp2);
printf("Files now closed");
exit(Ø);
```

FGETS()

This function is used to get a buffered line from a file. Its syntax is:

```
eofind = fgets( buf, max, fp );

buf        -  address of the buffer area.

max        -  the maximum length of the input string.

fp         -  the file pointer.
```

Fgets() is used to obtain a buffered line from a file. A file may be the console keyboard, the RS-232 interface, or any input device or disk file. Up to (max) bytes will be placed in the buffer. Input is terminated when either an end of line (ODH) or end of file is encountered or maximum buffer size is reached.

For compatibility with LDOS JCL files, keyboard line input is performed using the @KEYIN system call. Fgets() recognizes the BREAK key as the end of file from the keyboard.

RETURN CODE

The end-of-file indication, "eofind", is the return code. "eofind" is NULL (zero) if an end of file is encountered; otherwise, "eofind" is <buf>.

NOTE

According to K&R, the newline character (\n) should be included in the buffer returned by fgets(). Release 1.0 of LC stripped the newline character. Starting with release 1.1, LC has been brought into agreement with K&R. An option, GETNL, has been provided to force the removal of the newline from the buffer. If **#option GETNL OFF** is specified, then fgets() will strip the newline character. GETNL defaults to ON so as to include the newline character.

FOPEN()

This function is used to open a file/device. Its syntax is:

```
fp  = fopen( fspec, mode );

fspec      -  is the address of the file specification.

mode       -  the address of the access mode identifier:
              "r" or "R" = read; "w" or "W" = write;
              "a" or "A" = append.
```

Fopen() allows the programmer to initiate access to a file. Except for standard input, output and error files which are automatically opened, all files must be opened using fopen(). "fspec" points to a file specification string. "mode" points to a string defining the mode of access. Allowable modes are read, write or append. Only the first character of "mode" is checked, and that character may be upper or lower case.

The file pointer is used whenever access to the opened file is needed. If zero is returned, an error occurred during the open process.

RETURN CODE

The file pointer, "fp", is returned if no errors are detected in the open operation. "fp" will be set to NULL (zero) if an error is detected during the open operation.

WARNINGS

Opening the same file for both input and output with two or more calls to fopen() should NOT be done. If the file is accessed in this manner, it will create unpredictable results, possibly causing loss of file integrity.

EXAMPLE

```
getfile(fname)
    char *fname;
{   FILE *fp;
    if ((fp=fopen(fname,"r")) == NULL)
    {   printf("Open error - %-2Øs\n",fname);
        exit();
    }
    else return fp;
}
```

FPRINTF()

This function is used to create a formatted image for output to a device/file. Its syntax is:

```
retcod = fprintf( fp, control, arg1, arg2, ... );

fp          -  is a file pointer.

control     -  is a string as specified under PRINTF().

argn        -  are arguments as specified under PRINTF().
```

RETURN CODE

The "retcod" will be zero if no error was detected in the output operation. If an error was detected, then "retcod" will be EOF (-1).

FPUTS()

This function will output a string to a file/device. Its syntax is:

```
retcod = fputs( string, fp );

string     -  is the address of the string to be output.
```

Fputs() outputs to the file defined by "fp", all characters pointed to by "string", up to the first zero byte.

RETURN CODE

The "retcod" will be zero if no error was detected in the output operation. If an error was detected, then "retcod" will be equal to "EOF" (-1).

WARNINGS

Calling fputs() with an invalid file pointer can result in destruction of files or other havoc.

FREE()

This function frees memory allocated with alloc(). Its syntax is:

```
free( ptr );

ptr        -  address of the bottom of the memory block.
```

Free() is called when a memory block allocated to the program by the function alloc() is no longer needed, and the programmer wishes to free the memory space for later use. "ptr" points to the first (lowest) byte allocated to the program by alloc().

RETURN CODE

There is no return code.

WARNINGS

Calling free() with an address other than that obtained from a call to alloc() will cause unpredictable results, probably a program crash when the next alloc() occurs.

GETC()

This function is used to fetch (input) a character from a file/device. Its syntax is:

```
c = getc( fp );

fp        -  is the file pointer.
```

Getc() is used to input a single byte from a file. "fp" must be obtained from fopen() or be a standard file pointer. Any of the 256 possible binary codes may be input using getc(). An end of file code, "EOF" (-1), is returned if end of file is encountered.

RETURN CODE

The return code is the integer value of the character input from the file. If an end of file is encountered, then "EOF" (-1) is returned.

WARNINGS

The "fp" must be a valid file pointer or devastation may result. You are warned!

If "c" is to be stored before testing for end of file it must be stored in an integer variable. If not, the end of file value will be truncated and will remain undetected.

EXAMPLE

```
filecopy(fp)    /* copy a file to the standard output */
FILE *fp;
{
    int c;
    while ((c = getc(fp)) != EOF)
        if (c != putc(c,stdout))
            abort("Output file write error");
}
```

GETCHAR()

This function is used to get a character from standard input. Its syntax is:

```
c = getchar();

          - there are no parameters.
```

Getchar() inputs a single byte from the standard input. Getc() is used to perform the input.

RETURN CODE

The return code is the integer value of the character input from the file. If an end of file is encountered, then "EOF" (-1) is returned.

WARNINGS

If <c> is to be stored before testing for end of file, it must be stored in an integer variable, or the EOF value will be truncated to 255.

EXAMPLE

```
bytes = lines = Ø;
while( (c=getchar()) != eof)
{   putchar(c);
    ++bytes;
    if (c == eol) ++lines;
}
```

GETS()

This function fetches (inputs) a buffered line from standard input. Its syntax is:

```
eofind = gets( buffer );

buffer    -  is a pointer to an 81 byte buffer.
```

Gets() inputs a line up to 8Ø characters long from the standard input and places the line in memory starting at the address given by <buffer>. Fgets() is used to perform the input.

RETURN CODE

The end-of-file indication, "eofind", is the return code. "eofind" is NULL (Ø) if an end of file is encountered; otherwise, "eofind" is <buffer>.

WARNINGS

The "buffer" must be at least 81 characters long.

MOVE()

This function will copy a memory block in memory. Its syntax is:

move(pfrom, pto, len);
pfrom - the address of the block to be moved.
pto - the address of the block's new starting address.
len - the length of the block, in bytes.

This function will perform a nondestructive move of a memory block. That means that if the "pto" address is less than the "pfrom" address, the move will start from the beginning of the block. If the "pto" address is greater than the "pfrom" address, the move will start from the end of the block.

RETURN CODE

There is no return code.

WARNINGS

There is no checking on the magnitude of "len"; thus, a move() with an erroneous value for "len" could overwrite a critical portion of memory.

ISALPHA() ISDIGIT() ISLOWER() ISUPPER()

These functions are used to test a character. Their syntax is:

```
retcod = isalpha( char );

retcod = isdigit( char );

retcod = islower( char );

retcod = isupper( char );

char      -  is the character under test.
```

"isalpha" is used to determine if a "character" is an upper-case or lower-case alphabetic (<A-Z, a-z>). "isdigit()" is used to determine if a "character" is a digit in the range <Ø-9>. "islower()" is used to determine if a "character" is an alphabetic in the range <a-z>. "isupper()" is used to determine if a "character" is an alphabetic in the range <A-Z>.

RETURN CODE

Each function will return a TRUE (1) or FALSE (Ø) value based on the results of the test.

EXAMPLE

```
if ( isdigit(char))
     printf("Character is <Ø-9>\n");
else if ( islower(char))
     printf("Character is <a-z>\n");
else if ( isupper(char))
     printf("Character is <A-Z>\n");
else
     printf("Character is none of the above\n");
```

PRINTF()

This function creates a formatted image for standard output. Its syntax is:

```
printf( control, arg1, arg2, ...);

control   - is a string containing transparent printing
            characters and conversion specifications.

argn      - are arguments to be formatted for the output
            print image.
```

This function is used to create an output image to the standard output device. The specifications for formatting the output are determined by the character string, "control". This string will contain ordinary characters copied directly to the output image and/or specifications denoting the field conversions of all arguments. The conversion specifications take the form of:

%{-}{xxx}{.yyy}char

As can be noted, the specification is a sequence of sub-fields of which the percent sign (%) and the "char" are mandatory. The percent is an "escape" character signaling the start of the field specification. The "char" denotes the format of the output field image {binary, decimal, string, etc.}. The sub-field specifications are interpreted as follows:

% = the mandatory specification initiator,

- = specifies that the value will be left-justified within the print field image,

xxx = specifies the minimum width of the print field image,

.yyy = specifies the maximum number of string bytes to print,

char = the conversion character {b=binary, o=octal, d=decimal, x=hexadecimal, s=string, c=character, u=unsigned}.

Any portion of the control string which cannot be interpreted as a conversion specification field is considered to be transparent printing characters and will be passed directly to the print image.

EXAMPLE

```
printf("%d characters, %d lines were copied\n", bytes, lines);
```

PUTC()

This function is used to output a character to a file. Its syntax is:

```
cret = putc( c, fp );

c        -  is the character to be output.

fp       -  is the file pointer for the output file.
```

Putc() is used to output single characters to a file. "c" is any of the 256 possible character codes. If an integer value is passed it is left-truncated, so that only the least significant byte is output.

RETURN CODE

The return code, "cret", is the character passed in "c" if no errors are detected otherwise, it will be different from the character passed in "c".

WARNINGS

"fp" must be a valid file pointer obtained from fopen() or one of the standard I/O pointers (stdin, stdout, stderr) or destruction of files may occur.

EXAMPLE

```
if ( putc( c, fp ) != c )
     return(1);
else return(Ø);
```

PUTCHAR()

This function is used to write a character to standard output. Its syntax is:

```
cret = putchar( c );

c         -  is the character to be output.
```

Putchar() outputs the character "c" to the standard output file. Putc() is used to perform the output operation.

RETURN CODE

The return code, "cret" is the character passed in "c", if no errors are detected.

WARNINGS

There are no warnings.

PUTS()

This function is used to output a string to the standard output. Its syntax is:

```
retcod = puts( string );

string    -  is the address of the string to be output.
```

Puts() outputs <string> to the standard output file. All characters up to the first zero byte are output. If an error occurs during output, the value "EOF" (-1) is returned; otherwise, a zero is returned.

RETURN CODE

The "retcod" will be zero if no error was detected in the I/O operation. If an error was detected, then "retcod" will be set to "EOF" (-1).

WARNINGS

There are no warnings.

EXAMPLE

```
if (argc!=3)
{   puts("Format error:  compare file1 file2\n");
    exit();
}
```

SBRK()

This function is used to allocate a memory block. Its syntax is:

```
ptr = sbrk( nbytes );

nbytes    -  an unsigned integer number of bytes needed.
```

Sbrk() reserves memory for use by a program from the system memory pool. The memory allocated by sbrk() cannot be deallocated until the program finishes execution. Alloc() uses sbrk() to request blocks of memory as needed. If the memory requested will only be needed for part of the execution of the program, it is recommended that alloc() be used.

RETURN CODE

The return code, "ptr", is the address of the allocated block of memory if the sbrk() was successful. If not enough memory is available to satisfy the request, "ptr" is set to NULL (Ø).

WARNINGS

Only memory allocated by sbrk() or alloc() should be used by the programmer for dynamic space. File opens and closes, including standard files, use these functions for setting up File Control Areas (FCA's). These FCAs can be clobbered if the program accesses unauthorized memory.

TOLOWER() TOUPPER()

These functions are used to convert a character from one case to the opposite case. Their syntax is:

```
c = tolower( char );

c = toupper( char );


char      -  is the character under test.

c         -  is the converted result.
```

"tolower" is used to convert an upper-case character <'A' through 'Z'> to a lower-case alphabetic <'a' through 'z'>). "toupper" performs the opposite function; a lower case character is converted to upper case. Both functions affect only alphabetic characters; numbers, special symbols, etc., are returned unaltered.

RETURN CODE

Each function will return the converted character, as required.

INSTALLATION LIBRARY

The installation library is a collection of functions very specific to the particular machine LC is running on. Also included are functions not considered to be "standard" C functions These functions group themselves into STRING functions, PLOTTING functions, and CONTROL functions. All of the functions are contained in the library, IN/LIB (note: STRCAT, STRCMP, STRCPY, and STRLEN are documented under the string functions; however, since they are standard C, they are physically located in the LC/LIB library).

If your LC program is going to make use of any of the functions contained in the IN/LIB library, you will have to insert the compiler directive statement:

```
#option INLIB
```

in your C source program.

PRIMITIVE PLOTTING FUNCTIONS

The plotting functions support the block graphics mode available to the CRT screen. The installation library plotting functions include functions to control individual pixels (picture elements), as well as to create various line constructions. The primitive functions are used to turn on, turn off, or determine the status of any point (pixel) in the screen image. The syntax of these functions is:

```
retcod = pixel( funcod, x, y );

retcod = point( x, y );

reset ( x, y );

set( x, y );

funcod    -  specifies whether the pixel is reset (Ø),
             set (1), or pointed (2).

x         -  specifies the horizontal coordinate.

y         -  specifies the vertical coordinate.
```

The "point()" function will return the status of the pixel at the coordinate, x,y. A return code of one (1) indicates that the pixel is turned on (light) while a zero (Ø) indicates that the pixel is turned off (dark). If the pixel contains something other than a graphic character, a negative two (-2) is returned. A negative one (-1) indicates the point x,y is out of range.

The "reset()" function will turn off the specified pixel while the "set()" function turns on the pixel. Neither of these two functions provides a return code unless x or y is out of range.

The "pixel()" function can be used to point, reset, or set the pixel depending on the function code supplied as the argument. "pixel()" issues a return code only when the function code passed is indicative of "point" (2) or when an argument is invalid.

The plotting functions specified above refer to the arguments detailed as follows:

funcod

This function code specifies the operation to be performed on the pixel. It can be an integer value in the range <Ø-2>. If the argument passed is outside of this range, the return code will be negative three (-3) indicating

an invalid function code. These codes are used as follows:

Ø - Indicates the "reset" function which will turn off (make dark) the pixel.

1 - Indicates the "set" function which will turn on (make light) the pixel.

2 - Indicates the "point" function which will return the status of the specified pixel. The status will be zero (Ø) for reset, one (1) for set, negative one (-1) if x1,y1 is not in the CRT image, or negative two (-2) if the specified pixel does not contain a graphic character.

x or y

This integer value specifies the pixel position along the x-axis (horizontal) or y-axis (vertical). The value is a virtual pixel, which means that it does not have to be a position in the CRT image. However, where a line is being constructed, only that part of the line actually in the CRT image area will be plotted. The direction away from the origin is always considered to be in the positive direction (for more information on this subject, see the pmode() function).-

RETURN CODE

Return codes are provided by the above functions where applicable. These codes are indicative of the following:

Ø - Indicates that the pixel is reset [point(x,y) or pixel(2,x,y)].

1 - Indicates that the pixel is set [point(x,y) or pixel(2,x,y)].

-1 - Indicates that the point x1,y1 is out of range (i.e. virtual and does not appear in the CRT image).

-2 - Indicates that the pixel does not contain a graphic character [point(x,y) or pixel(2,x,y)].

-3 - Indicates that the function code passed to pixel() is invalid (not in the range <Ø-2>).

EXAMPLE

This routine plots a horizontal line:

```
for (x = Ø, y = 4Ø; x < 128; x++)
     set( x, y );
```

ADVANCED PLOTTING FUNCTIONS

These functions are used to plot geometric shapes {lines, rectangles, and circles}. Their syntax is as follows:

```
retcod = box( funcod, x1, y1, x2, y2 );

retcod = circle( funcod, x1, y1, r1 );

retcod = line( funcod, x1, y1, x2, y2 );

funcod    - an operation code to set (1) or reset (Ø) the
            pixels involved in the geometric plot.

x1,y1     - the coordinate of the first point defining
            the geometric shape.

x2,y2     - the coordinate of the second point defining
            the geometric shape.

r1        - the radius of the circle in "y" units.
```

The "line()" function will plot a line connecting coordinate point x1,y1 with coordinate point x2,y2.

The "circle()" function will plot a circle at coordinate center point x1,y1 of radius r1. The integer value, "r1", specifies the radius of the circle. Since block graphics are generally taller than their width, it is necessary to specify the radius in units of either "x" or "y". Within these plotting functions, "r1" is a value representing the radius in "y" units.

The "box()" function will plot a rectangle around the diagonal specified by the coordinate point pairs, x1,y1 and x2,y2. If the coordinates specified either equal x (x1 = x2) or equal y (y1 = y2), then the rectangle will diminish to a line. The rectangle will collapse to a point if both x1=x2 and y1=y2.

VIRTUAL POINTS

The concept of virtual points is an important one. What it means is that your plotting routines do NOT have to limit themselves to the CRT image area. For example, a circle(1,Ø,Ø,2Ø); function describes a circle about the origin. This means that a portion of the circle would be plotted off of the CRT image. The plotting functions permit your arguments to describe such "virtual" images; however, any portion of the geometric shape that would be outside of the CRT image area is inhibited. Thus, in the above example, only a portion of a circle (an arc) will be plotted.

It is also important to note that if any virtual pixel is described in your arguments, the function will return a negative one (-1) after completing the entire geometric plot. Your program can make use of this return code if it needs to detect this fact.

RETURN CODES

Return codes are provided by the above functions where applicable. These codes are indicative of the following:

-1 - Indicates that the coordinate points x1,y1 , x2,y2, or a portion of any plot is out of range (i.e. virtual and does not appear in the CRT image).

-3 - Will be returned if the function code passed is invalid (not in the range <0-1>.

EXAMPLE

The following routine will plot increasing rectangles starting at the center of the CRT image:

```
#define X 63 /* 64 x 16 screen size */
#define Y 23
#define X 79 /* 80 x 24 screen size */
#define Y 35
for (x1=X,y1=Y,x2=X+1,y2=Y+1; x1 >= 0; x1--,y1--,x2++,y2++)
     box(1, x1, y1, x2, y2 );
```

Try out the next example program:

```
#option INLIB
main()
{
   int x1,x2,y1,y2,t,t1;
   for ( x1=0, y1=0, x2=127, t = 0 ; t <= 47 ; t++ )
      {  line(1,x1,y1,x2,t);
         line(0,x1,y1,x2,t);
      }
   for ( y2=47,t = 127 ; t >= 0 ; t-- )
      {  line(1,x1,y1,t,y2);
         line(0,x1,y1,t,y2);
       }
   exit;
}
```

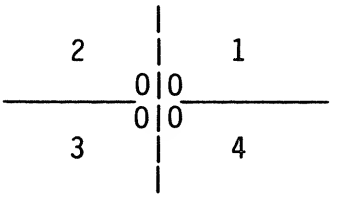
PLOTTING CONTROL FUNCTIONS

Two functions are provided that interface with and control certain aspects of the pixel plotting functions. The "pmode()" function establishes the CRT image area as one of the four quadrants in the cartesian coordinate system. Another function, "ploc()", establishes the starting address of the CRT image area. The syntax of these functions is:

```
ploc( address );

retcod = pmode( quadrant );

address   -  specifies the starting address of the
             plotting image area. Plotting functions use
             the CRT address unless changed by pmode().

quadrant  -  sets the plotting image to quadrant <1-4> of
             the x-y plane {initialized to quadrant 4}.
             If quadrant = Ø, then the current quadrant
             number in effect will be returned.
```

The pmode() function is quite useful when your application concerns the graphing of mathematical functions in the standard cartesian coordinate system. Since most functions are graphed in the first quadrant, a "pmode(1)" will establish the image area for that purpose. Please note that any characters/graphics currently on the screen at the time the pmode() is given are left undisturbed - pmode() does NOT refresh the current screen contents to the revised quadrant but prepares the plotting functions for the new quadrant.

"Quadrant" is used when changing the base origin of the plot image area with the pmode() function. The image area is considered to represent only one quadrant of the x-y plane in the cartesian coordinate system. The quadrants are numbered as follows:



with the point Ø,Ø (the origin) appearing at the corner identified with the letter "O". The standard quadrant used by the plotting functions will be quadrant 4 unless changed with a pmode() function call. Remember that the direction away from the origin is always considered to be positive.

The ploc() function can be very powerful in creating dynamic displays. By establishing an off-CRT buffer equal in length to the CRT image area, its

address can be passed via ploc() so that the plotting functions plot into the buffer. The buffer could be subsequently moved to the CRT image area with the move() function.

On machines with memory-mapped video, "address" is initialized to the video memory address (i.e. 0x3c00 on a Model I or Model III). Thus, if you pass some other address via ploc(), you can restore plotting to the CRT by passing this CRT memory address via another ploc() invocation.

On machines which do not have user-accessible memory-mapped video, an address of zero is used to reference the CRT (using PRO-LC on the Model 4, for instance). Passing a zero as the address will also return the current ploc() address as the function value.

RETURN CODES

Return codes are provided by the pmode() function where applicable. These codes are indicative of the following:

1-4 - Indicates the current quadrant in effect when a pmode(0); function is invoked.

-1 - Returned if "quadrant" is not in the range <0-4>.

STRING FUNCTIONS

The most important bit of information to convey at this point concerning the use of strings and the C language, is that the language provides no internal mechanics for dynamic string maintenance. Strings are generally stored in character arrays - arrays are fixed in length at the time of their declaration. Therefore, when you employ the string functions contained in this installation library, remember that your application must provide the proper array sizes to deal with the expected lengths of the strings. Where string lengths are indeterminate at the time the application is coded but are determined rather at run time, it may be prudent to consider testing the length of a string operation result prior to actually performing the intended operation to ensure that the operation will not exceed the array size of the array receiving the string result.

Another point worth remembering is that there is no upper limit on the length of a string in the C language. A string is stored contiguously in memory. The last character of the string is denoted by a null byte (hex zero) at the end. Thus, any array used to store a string should be defined with a size one byte greater in length than the maximum length of the string it is to contain. This will allow for the storage of the terminating zero byte. For instance, the string "Hello", is stored as (shown in hex):

48 65 6C 6C 6F ØØ

The string functions provided in the installation library correlate with the string functions provided in various implementations of Microsoft BASIC. Again, remember that no dynamic string allocation/compression takes place in these routines.

The string functions, strcat(), strcmp(), strcpy(), and strlen() are documented in this section. However, since these functions are considered to be "standard C" functions, the routines are supplied in the standard library, LC/LIB.

The functions provided take the following format:

```
strcat( dest, source );

retcod = strcmp( string_1, string_2 );

strcpy( dest, source );

retcod = strepl( source, dest, pos, count );

strept( dest, source, repeat );

retcod = strfind( dest, source, pos );

stright( dest, source, count );

strleft( dest, source, count );

retcod = strlen( source );

retcod = strmid( dest, source, pos, count);

count     -  is the integer sub-string length.

dest      -  is a pointer to the destination string.

pos       -  starting index position or array subscript.

source    -  is a pointer to the source string.

repeat    -  a repetition counter.
```

STRCAT()

The strcat() function will concatenate (append) the source string to the destination string.

STRCMP()

The function, strcmp(), will compare string_1 to string_2. If the string_1 would appear above string_2 in an ascending sorted list, the return code will be negative. (<Ø). If the two strings are equal, a zero (Ø) will be returned. A return code of a positive value greater than zero (>Ø) indicates string_1 to be below string_2 in an ordered list. If you are unfamiliar with how ASCII strings are "ordered", perhaps a strong example will clarify this discussion. The following is an ordered list of strings in ascending order:

```
a8bcde
abc
abcd
jim
karl
rich
roy
this_is_a_long_string
```

Keep this ordered list in mind in the following examples. The statement:

```
if (strcmp("abc","a8bcd") < Ø) ? printf("above") : printf("below");
```

should print the word, "below" since the string, "abc" is below the string "a8bcd" in an ascendingly sorted list. The statement:

```
if (strcmp("abc","abcd") < Ø) ? printf("above") : printf("below");
```

should print the word, "above" since the string, "abc" is above the string "abcd" in an ascendingly sorted list.

STRCPY()

The strcpy() function copies an image of the source string to the destination string buffer.

STREPL()

The function, strepl(), replaces that portion of the destination string starting at relative position "pos" and continuing for "count" characters [the destination substring] with the source string. The arguments "pos" and "count" control where and how much of the destination string is to be replaced (i.e. what is the substring). The length of the replacement string is the length of the source string. If "count" is zero (Ø), then an insert operation is performed without deleting any characters of the destination string. If the source string is null (i.e. of zero length), then only the identified sub-string is deleted. If "dest+pos" exceeds the bounds of the destination string, an out-of-range error will be returned and the string operation will be aborted. Bear in mind that this function behaves differently than the Microsoft BASIC MID$= function; however, the LC strepl() function performs as a string replacement algorithm should perform.

STREPT()

The strept() function replicates the source string into the destination string the number of times indicated by "count". Note that the replication uses the entire source string and not just the first character. Thus a statement such as:

```
strept(newstring,"*.",1Ø);
```

will create newstring equal to "*.*.*.*.*.*.*.*.*.*.*.".

STRFIND()

The function, strfind(), will search the destination string for the first appearance of the source string. The destination string will be searched starting at the position "dest+pos". If the source string is a null string, the value of "pos" will be returned. If the destination string is a null string, a negative one (-1) will be returned. If the source string is found in the destination string, its position relative to the beginning of the source string will be returned. If the source string is not found (i.e. is not a sub-string of the destination), a negative one (-1) will be returned.

STRIGHT()

The stright() function will copy the rightmost "count" characters of the source string (the sub-string) to the destination string. This is NOT an append operation. The destination string is replaced with the sub-string. If "count" is zero, the destination becomes a null string. If the "count" is greater than the source string length, the entire source string is copied.

STRLEFT()

The function, strleft(), will replace the destination string with the leftmost "count" characters of the source string. If count is zero, the destination becomes a null string. If the "count" is greater than the source string length, the entire source string is copied.

STRLEN()

The strlen() function returns the length of the source string.

STRMID()

The function, strmid(), will replace the destination string with the substring of "count" characters starting at position "source+pos" of the source string. If "count" is zero, the destination string will be null. Also, if "source+pos" exceeds the bounds of the source string, an out-of-range error (-1) will be returned and no string replacement will occur. "Count" may be greater than the length of "source" plus "pos".

STRING FUNCTION ARGUMENTS

The string functions specified above reference various arguments detailed as follows:

DEST

This represents a pointer to a character array. The argument that is passed to a function is an address when the argument references an array; therefore, a pointer is identified to the string functions by the very nature of the character array declaration as in:

```
char s[81]
```

which establishes a character array capable of holding up to an 8Ø-character string.

SOURCE

This also represents a pointer to a string which is used as the "source" string where the function requires more than one string in its arguments.

POS

This represents a starting position relative to the beginning of a string. It is essentially used as an index or subscript into the character array (Ø <= POS < n).

COUNT

This parameter is used, where required, to indicate the length of some substring. In the case of strept(), it is used to specify the replication quantity.

RETURN CODES

The return code of strcmp() is <Ø, Ø, or >Ø as noted above. The return code of strlen() is the length of the target string. The functions strmid() and strepl() will return a negative one (-1) if the arguments specify a resulting position outside the range of the string. For instance:

```
if ( strmid(s,"error",6,3) == -1 ) printf("String error!");
```

will result in the error message display since the position, 6, is not in the range of the string, "error". The remaining string functions do not have return codes.

CONTROL FUNCTIONS

The remaining functions included in the installation library not previously detailed perform miscellaneous tasks. The syntax of these functions is as follows:

```
fill( buffer, count, char );      - zap a memory block
retcod = freemem();               - fetch memory size
retcod = inkey();                 - scan the keyboard
retcod = inport( port );          - input from a port
outport( port, value );           - output to a port
retcod = curpos();                - fetch cursor position
cursor( col, row );               - reposition the cursor
date( s );                        - fetch the system date
time( s );                        - fetch the system time
cmdi( "command string" );         - exit & command DOS
retcod=cmd( "command string" );   - command DOS & return
retcod=call( address, regs );     - generalized call
```

FILL()

The function, fill(), will propagate the character, "char", into the memory "buffer" for "count" bytes. If "char" is passed as an integer value, the low-order byte is used for the propagation. Note the difference between strept() and fill().

INKEY()

The inkey() function makes a single scan of the keyboard and returns the ASCII value of any depressed key. It will return a zero if no key is pressed.

INPORT

The function, inport(), returns as an integer, the value read from the specified port.

OUTPORT()

The outport() function outputs the integer value to the port. The value is truncated to its low-order byte.

CURPOS()

The current location of the cursor can be recovered with the curpos() function. It returns the cursor position as an encoded value. The cursor row is in the high-order byte while the cursor column occupies the low-order byte of the integer return code.

CURSOR()

To reposition the cursor, use the cursor() function. The cursor is re-positioned to the location identified by the arguments. If the position that would result is not on the CRT screen, a range error (-1) is returned. "x" must be in the range, <0-X>, while "y" must be in the range, <0-Y>. For 64 x 16 screen sizes, the X & Y limits are 63 & 15. For 80 x 24 screen sizes, the X & Y upper limits are 79 & 23 respectively.

DATE()

The date() function will place the system date into the string "s". The format is MM/DD/YY. The string should be defined as a character array of minimum dimension 9.

TIME()

The time() function will place the system time into the string "s". The format is HH:MM:SS. The string should be defined as a character array of minimum dimension 9.

CMDI()

The function, cmdi(), will invoke an exit from the running LC program and schedule the DOS execution of the command contained in string "command string". This could be used, for instance, to chain to another C program. The cmdi() argument can be either a string defined in the function call or a pointer to a character array which contains the command string.

CMD()

If you want to execute a command and return to your LC program, use the cmd() function. This function will pass the command stored in string "command string" (or the string pointed to by a pointer argument) to the DOS command

interpreter. Upon completion of the command, control will be returned to the running LC program. Your program and variables will be saved during the execution of the command. If the executing command returns through @EXIT, a return code of zero (0) will be retrieved. If the @ABORT exit is taken, the return code generated will be obtained from the value contained in register pair "HL". If this value is positive (i.e. bit 15 reset), it becomes the return code. If the value is negative (i.e. bit 15 set), then a negative one (-1) will be returned.

CALL()

A generalized assembly language interface routine, call(), is available in the installation library. "Regs" is an integer array of dimension 6 which should contain the quantities you want placed into the register pairs {AF, BC, DE, HL, IX, and IY for regs[0]-regs[5] respectively} prior to calling the routine at location "address". The "regs" array will be stuffed with the register contents that existed upon return from the called routine. The return code will be zero (0) if the Z-flag is set upon return from the called routine; otherwise, the return code is one (1). For more information on the use of call(), see the chapter on ADVANCED TOPICS.

On SuperVisor Call accessible systems, an "address" value of less than 256 will be interpreted as an SVC reference in lieu of a CALL address. The call() function will then use the value passed in "A" of regs[0] as the SVC number.

FREEMEM()

Freemem() returns the maximum amount of memory which can be obtained from alloc() or sbrk().

LC FLOATING POINT LIBRARY

LC does not have floating point arithmetic built into the compiler. Thus, floating point expressions are not allowed in the normal manner. However, this floating point function library allows the programmer to use the floating point routines built into the TRS-8Ø BASIC ROM. These functions provide access to single and double precision math, all the trigonometric functions, random number generation, and conversion to and from ASCII strings.

Before any floating point math can be done, the function "fpinit()" must be called. Fpinit() initializes some data areas used by the TRS-8Ø ROM floating point routines and sets up linkages for error recovery. The calling of fpinit() and the automatic search of the floating point library is accomplished when you add the compiler declaration:

```
#option FPLIB
```

to your C-language source program. This establishes the protocol necessary to invoke an automatic search of the floating point library in the LC/ASM file via the "*SEARCH FP/LIB" statement. The floating point initialization function, fpinit(), is normally called automatically by LC's initialization routines if "#option FPLIB" has been specified. However, if the user changes or substitutes different initialization code, the user's program must call fpinit().

Numbers may be stored in two different formats: single or double precision. Throughout the library routines two consistent abbreviations are used: "f" for floating point SINGLE precision, and "d" for floating point DOUBLE Precision. Functions beginning with or containing either of these abbreviations operate on the precision indicated. Single precision numbers are stored in four bytes; double precision numbers in eight bytes. The programmer may declare either an integer array, or a character array to allocate space for variables in their program. Both of the statements,

```
char fpnum[4];
int fpnum[2];
```

declare a 4-byte single precision field. Both of the statements,

```
char dpnum[8];
int dpnum[4];
```

declare an 8-byte double precision field.

Take care in how you pass the parameters required by each function: THE ADDRESS OF THE FIELDS ARE PASSED. Thus, if the declarations, "char fpnum[4],dpnum[8];", are used when calling a floating point library routine, the address of the first character (represented by the array name) should be passed. For example:

```
atof("3.1416",fpnum);
```

```
atod("3.1415926",dpnum);
```

converts the strings containing PI to single and double precision and places the results into memory at locations fpnum and dpnum, respectively.

If you are writing an assembly language routine that will interface to the FP/LIB routines, the assembler statement:

```
@_FPLIB   DEFL   -1
```

should appear in your code. This statement forces a search of FP/LIB in the LC/ASM file.

We wish to acknowledge our indebtedness to Insiders Software Consultants, Inc. for their book, THE BØØK - ACCESSING THE TRS-8Ø ROM, Volume 1, which provides vital information on interfacing to the ROM math routines. Although the functions in FPLIB do not require any knowledge of how the BASIC ROM in the TRS-8Ø functions, it can be helpful to have "THE BØØK" for a reference to explain the details of floating point operation.

FPINIT

This function is used to initialize interfacing to the ROM math routines and force a search of the floating point library during the assembly of LC/ASM. Its syntax is:

```
fpinit();

          -  there are no arguments.
```

This function is normally called automatically by an LC program which has any module specifying "#option FPLIB". In cases where normal initialization has been bypassed, a direct call to fpinit() by the user program will be necessary since fpinit() MUST BE CALLED before using any of the floating point functions in the library. It sets up certain data areas used by the floating point ROM routines, and provides error recovery linkage.

SINGLE PRECISION OPERATIONS

The following functions can be used to operate on single precision fields (length 4). Their syntax is as follows:

retcod = fadd(v1, v2);	-	v1 = v1 + v2
retcod = fsub(v1, v2);	-	v1 = v1 - v2
retcod = fmul(v1, v2);	-	v1 = v1 * v2
retcod = fdiv(v1, v2);	-	v1 = v1 / v2
retcod = fabs(v1, v2);	-	v1 = abs(v2)
retcod = fatn(v1, v2);	-	v1 = arctan(v2)
retcod = fcmp(v1, v2);	-	compare v1 to v2
retcod = fcos(v1, v2);	-	v1 = cos(v2)
retcod = fexp(v1, v2);	-	v1 = exp(v2)
retcod = ffix(v1, v2);	-	v1 = fix(v2)
retcod = fint(v1, v2);	-	v1 = int(v2)
retcod = flog(v1, v2);	-	v1 = log(v2)
retcod = fraise(v1, v2);	-	v1 = v1 ** v2
retcod = frnd(v1, v2);	-	v1 = rnd(v2)
retcod = fsgn(v1);	-	retcod = sgn(v1)
retcod = fsin(v1, v2);	-	v1 = sin(v2)
retcod = fsqr(v1, v2);	-	v1 = sqr(v2)
retcod = ftan(v1, v2);	-	v1 = tan(v2)
v1 & v2 - are the address of single precision fields.		

These functions perform calculations on two single precision fields and place the result in the first field specified in the argument list (v1). In the event of an error, v1 is unchanged and can be examined to determine the cause of the error. Only SINGLE PRECISION variables can be handled properly by these functions. Use the format conversion functions described later in this section to derive the proper precision.

RESULTS

In all of these functions, v1 will contain the result of the calculation if no error is detected. Any error (such as overflow, underflow, etc.) will leave the v1 argument unchanged. The argument "v2" is not altered in any way by the functions.

RETURN CODES

Each function has a return code of zero (Ø) if no errors occurred during the operation; otherwise, it will return a LEVEL II BASIC error code. The fcmp() function returns -1, Ø, or +1 depending on whether v1 is less than, equal to, or greater than v2. Typical error codes would be:

2 - Syntax error
5 - Illegal function call [log(negative number)]
6 - Overflow
11 - Division by zero

DOUBLE PRECISION OPERATIONS

The following functions can be used to operate on double precision fields (length 8). Their syntax is as follows:

retcod = dadd(v1, v2);	- v1 = v1 + v2
retcod = dsub(v1, v2);	- v1 = v1 - v2
retcod = dmul(v1, v2);	- v1 = v1 * v2
retcod = ddiv(v1, v2);	- v1 = v1 / v2
retcod = dabs(v1, v2);	- v1 = abs(v2)
retcod = dcmp(v1, v2);	- compare v1 to v2
retcod = dfix(v1, v2);	- v1 = fix(v2)
retcod = dint(v1, v2);	- v1 = int(v2)
retcod = dsgn(v1);	- retcod = sgn(v1)
v1 & v2 - are the address of double precision fields.	

These functions perform calculations on two double precision fields and place the result in the first field specified in the argument list (v1). In the event of an error, v1 is unchanged and can be examined to determine the cause of the error. Only DOUBLE PRECISION variables can be handled properly by these functions. Use the format conversion functions described later in this section to derive the proper precision.

RESULTS

In all of these functions, v1 will contain the result of the calculation if no error is detected. Any error (such as overflow, underflow, etc.) will leave the v1 argument unchanged. The argument "v2" is not altered in any way by the functions.

RETURN CODES

Each function has a return code of zero (Ø) if no errors occurred during the operation; otherwise, it will return a LEVEL II BASIC error code as described in the list under SINGLE PRECISION. The dcmp() function returns -1, Ø, or +1 depending on whether v1 is less than, equal to, or greater than v2.

DATA CONVERSION FUNCTIONS

The need will arise to convert between double precision, single precision, integer, and ASCII string. The following functions exist for this purpose:

```
retcod = atod( str, dvar );      ASCII to double
retcod = atof( str, fvar );      ASCII to single
retcod = dtoa( dvar, str );      double to ASCII
intval = dtoi( fvar );           double to integer
retcod = dtof( dvar, fvar );     double to single
retcod = ftoa( fvar, str );      single to ASCII
retcod = ftod( fvar, dvar );     single to double
retcod = itod( ivar, dvar );     integer to double
retcod = itof( ivar, fvar );     integer to single
intval = ftoi( fvar );           single to integer
dvar      - specifies a double precision field.
fvar      - specifies a single precision field.
intval    - specifies the integer value returned.
ivar      - specifies an integer value.
str       - specifies a character string field.
```

The above conversions should be self-explanatory. The conversions not shown above that convert ASCII to integer and vice versa, are part of the standard library and are documented in a preceeding section of the LC LIBRARIES.

The C itoi() function has not been implemented, for obvious reasons.

FLOATING POINT EXAMPLE

The following illustrates how the floating point library can be used in an application. The example is derived from K&R, page 8.

```
/* fctab - print Fahrenheit-Celsius table
    for f = -8, 12, 32, ..., 312 */
#include stdio/csh        /* include standard header file */
#option fplib             /* force search of FPLIB */
main()
{
    int lower, upper, step, fahr;
    char celsius[4];      /* provide space for "float" */
    char fivedivnine[4];  /* space to hold (5.Ø/9.Ø) */
    char temp[4];         /* temporary work space */
    char thirtytwo[4];    /* space to hold 32.Ø */
    char celsius_str[8];  /* space for ASCII result */

    lower = -8;  /* lower limit of temperature table */
    upper = 312; /* upper limit */
    step = 2Ø;   /* step size */
/*
    Note that the calculation (5.Ø/9.Ø) was removed from the
    body of the "while" loop to speed up calculations!
*/
    atof("5.Ø",fivedivnine);    /* float 5 */
    atof("9.Ø",temp);           /* float 9 */
    fdiv(fivedivnine,temp);     /* calc 5.Ø/9.Ø */

    atof("32.Ø",thirtytwo);     /* float 32.Ø */
    fahr = lower;               /* initialize to starting value */

    while ( fahr <= upper )     /* "fahr" & "upper" are integers */
    {
        itof(fahr,celsius);          /* float fahr */
        fsub(celsius,thirtytwo);     /* fahr - 32.Ø */
        fmul(celsius,fivedivnine);   /* (5.Ø/9/Ø)*(fahr-32.Ø) */
        ftoa(celsius,celsius_str);   /* result to ASCII */
        printf("%-6.3d        %-8.8s\n",fahr,celsius_str);
        fahr += step;    /* note the assignment operator */
    }
}
```

UTILIZING ASSEMBLY-TIME OPTIONS

LC provides certain options which can help the programmer to generate efficient programs. These options can be specified from the C source code by using the #option statement. The defaults to these options are set in the file, LCMACS/ASM, so that no options need be set, except when the default is not the desired option. Since the #option statement generates a DEFL statement in the assembly language source output, it can also be used to control options in any user libraries or separately compiled modules. If used to control options in separately compiled modules, the #option statement must be in a module which precedes the module whose option is to be specified.

Bear in mind that an option, once set, can be changed by another #option statement. Thus, when assembling separately compiled modules, care should be taken that a later module will not change the option set in the current module, unless this is desired.

Options which are switches can be set to ON (-1) or OFF (0). These specify whether or not some option is to be active. The constants ON and OFF are defined in STDIO/CSH. Options which are not switches may be set to the value desired by specifying the value in the #option statement.

ARGS

This option controls the generation of argc and argv, the command line argument parsing. If the option is turned ON, the arguments are created and placed on the stack so that the user may access them from main(). If turned OFF, some savings in memory result, as no parsing of the command line is done. ARGS defaults to ON. For more information on argc and argv, see K&R.

ERRORMSG

If set OFF, this option suppresses the DOS I/O error message display.

FIXBUFS

This option, if set to ON, will cause standard I/O to pre-allocate all buffers needed for standard I/O. This eliminates the need for the dynamic memory functions to be loaded, and prevents users from locking themselves out from further fopen()'s by allocating all of memory. This also guarantees that the standard I/O files can always be opened if the program is loaded. FIXBUFS defaults to OFF.

FPLIB

This option is specified if you are referencing any function contained in the floating point library, FP/LIB. FPLIB defaults to OFF.

GETNL

This option concerns the inclusion of the newline character in the fgets() function. See page 4-6 for its use.

INLIB

This option is specified if you are referencing any function contained in the installation library, IN/LIB. INLIB defaults to OFF.

KBECHO

In certain types of programs, such as screen and graphics editors, the programmer may choose to disable the echoing of characters typed at the keyboard. However, for most programs it is desirable to be able to see what is being typed, even if standard output has been redirected. This option allows this flexibility. When ON, KBECHO will cause the getc() function to echo all characters input from the keyboard to the video. This holds true for ANY file opened as "*KI", not just the standard input. KBECHO defaults to ON.

MAXFILES

This option requires a numeric argument instead of OFF or ON, as it is not a switch. The number will be used to set up control storage for user files. The maximum number of files which can be opened at the same time is controlled by MAXFILES. The standard I/O files are not counted in this number. For example, if no user files were needed, then MAXFILES could be set to zero. MAXFILES also controls the number of buffers pre-allocated when the FIXBUFS option is ON. Thus, it is best to use MAXFILES when FIXBUFS is specified to minimize the memory pre-allocated for file buffers. MAXFILES defaults to allow the user eight (8) concurrently opened files.

REDIRECT

The REDIRECT switch controls the I/O redirection feature of the standard library. When REDIRECT is ON, standard files can be redirected by command line specification. When OFF, no redirection processing takes place, and the standard I/O files are set up as permanently attached to the keyboard and screen. The REDIRECT switch overrides the ARGS switch; if REDIRECT is ON, ARGS is also forced to ON. REDIRECT defaults to ON.

ZVAR

The ZVAR switch can be used to invoke the initialization of all variables to zero. If ZVAR is not optioned, only space will be reserved for variables and their initial values will be undefined. Note that this switch option may be turned off or on throughout the program.

SEPARATE COMPILATION

LC supports separate compilation: functions and modules can be compiled at different times, then assembled together to produce one program. This facilitates the creation of compiled function libraries, and results in great time savings. Commonly used functions can be compiled once, then only assembled into new programs, without recompiling. Large programs may be segmented and each segment compiled separately, then assembled as a whole. With the "extern" and "static" statements, the variables used in a module may be specified as external or local.

When separately compiling modules which reference variables in other modules, two approaches may be taken to supply declarations for the shared variables. The "-global" option of LC may be turned off and on, so that only one module actually defines the space for global variables. The other method, which is the proper method and results in better structure in programs, is to define variables as extern when referenced by all but one of the modules.

USING THE -GLOBAL OPTION OF LC

All shared global declarations should be placed in a separate file, to be #include'd by all modules which use them. Only one module can have +Global specified when compiled; all other modules must be compiled with -Globals in the LC command line. It is usually convenient to use +Global when compiling the module containing main(). This method is not normally recommended. It is only supplied as a convenience to those attempting to compile extremely large programs already written without using extern and static. It is best to learn to structure your functions into "units": modules containing a set of related functions and their related internal and shared variables.

USING EXTERN AND STATIC

When writing a large program, it is best to try and logically structure your program into modules containing related functions with the data structures they use within the same module. Any data structures or functions in a module which need not be accessed by any external function can be declared as "static". These static functions and variables will be unique in name when assembled, and will not be accessible to other modules, so there will be no conflicts in naming. Those data structures and functions declared in the module which need to be accessed by functions in other modules should be declared without any storage class. This causes these functions and data structures to become "external", meaning that they are defined in this module, and can be accessed from other modules. When using a function or data structure declared in another module, the "extern" statement is used to declare the type of the object. "extern" is required for accessing variables outside the module. However, a function may be used without an "extern"; the compiler will assume that the function returns an integer value. If any other result is returned, the function must be declared "extern".

ASSEMBLING SEPARATELY COMPILED MODULES

Each module must be read and assembled by EDAS in the same assembly. The *GET assembler directive is used for this purpose. A file should be created with *GET statements for all modules to be included in a program. This file name is then specified as the program name when assembling when using LC/ASM. Here is an example of the GET file

(listing of MYPROG/ASM)

```
*GET MYMAIN
*GET MYFUNCS
*GET MORECODE
```

When assembling, load LC/ASM into EDAS and then issue the following <C>hange command:

C/CPROGRAM/MYPROG/

to set up the *GET filename. Then assemble as you would normally.

CREATING USER LIBRARIES

We encourage LC users to create libraries of commonly used functions. This increases your productivity, since functions need not be rewritten for each program. A library should contain functions which are self-contained; i.e, they do not require the calling function to know about the library module's internal structure, and do not assume anything about data structures that the calling function declares. In structured programming lingo, library functions should be data-coupled and functionally cohesive. Also, functions should be tested and be well debugged before being placed in a function library.

IN-LINE LIBRARIES

A user library can be created by appending LC output (assembly source) files together. The assembly of each function in the library can be controlled by the IFREF pseudo-op available in EDAS. When you reference a user library function in your program, the subsequent assembly of the program will cause the IFREF to be true for that function. Otherwise, if you have not referenced (called) that function, IFREF is false, so the function is not assembled. This method is simple and does not require the use of the PDS utility. Its disadvantage is that the entire library must be read by EDAS even if some of the modules do not get assembled. The *SEARCH directive for searching PDS libraries in EDAS is much faster for larger libraries.

The creation of an in-line (one after another) library should go something like this: The user types the following commands:

```
 EDAS    YOU TYPE
------  ------------------------------------------------------
 >      I <ENTER>              (NOW IN INSERT MODE)
ØØ1ØØ   IFREF FUNCA
ØØ11Ø   <BREAK>                (EXIT INPUT MODE)
 >      LFUNCA                 (LOAD THE FIRST FUNCTION)
 >      IB                     (INSERT TO BEGIN AFTER LAST)
Ø243Ø   ENDIF
Ø244Ø   IFREF FUNCB
Ø245Ø   <BREAK>                (BREAK TO EXIT INPUT MODE)
 >      LFUNCB                 (LOAD THE SECOND FUNCTION)
 >      IB
Ø441Ø   ENDIF
Ø442Ø   <BREAK>                (HIT BREAK TO EXIT INPUT MODE)
 >      W USRLIB:1             (SAVE THE LIBRARY ON DRIVE 1)
```

Any number of functions can be added in this fashion, by appending to the existing library. Each function could also be edited separately and then appended together with the LDOS APPEND command (use the STRIP parameter of APPEND to remove the EOF byte (X'1A') from "library" when you are appending another module). This process generates a library which looks like this:

```
     IFREF FUNCA
(FIRST FUNCTION)
     ENDIF
     IFREF FUNCB
(SECOND FUNCTION)
     ENDIF
(AND SO ON...)
```

A different approach is needed when several functions have been compiled together in the same LC invocation. A temporary label is needed to control the invocation of the module. This type of construct looks like this:

```
$INVOKE_IT DEFL Ø      ;Default to no invocation
           IFREF FUNCA
$INVOKE_IT DEFL -1     ;Get it
           ENDIF
           IFREF FUNCB
$INVOKE_IT DEFL -1     ;Get it
           ENDIF
           IFREF FUNCC
$INVOKE_IT DEFL -1     ;Get it
           ENDIF
;*=*=*
;    Now invoke the module if needed
;*=*=*
           IF $INVOKE_IT
( THE MODULE GOES HERE )
           ENDIF
```

With the construct shown above, the module will be assembled if any of the functions are referenced. It is better to separate the functions into separate modules if possible.

An in-line library is searched by EDAS sequentially from beginning to end. Therefore the order in which functions are placed in the library becomes important. If a function in the library is called by another function within the library, then the calling function must appear first. This is because the IFREF for the called function will not be true until the calling function is assembled. So the general rule is: Calling functions first, called functions last.

PDS LIBRARIES

The libraries which come with the LC compiler are actually Partitioned Data Sets (PDS). A PDS is a file which is partitioned into individual members. The Partitioned Data Set (PDS) utility, available from MISOSYS, is

used to create and maintain PDS libraries. PDS members may be executable commands, data files, source files, etc. In the case of the LC libraries each PDS member is an assembly source module. EDAS searches PDS libraries by performing an IFREF check on each member name. Only those members which have been referenced but not as yet defined are read and assembled by EDAS. For the rest of this discussion we shall assume the reader has a working knowledge of the PDS utility.

The PDS(BUILD) command is used to create and initialize a PDS library. Once created, modules can be appended to the library using the PDS(APPEND) command. The function name in a module is limited to eight characters, with no underline allowed. When the module contains only one function, it is convenient to have the module name be the same as the function name. In this case the command:

pds(append) myfunc/asm my/lib.pds

is sufficient to add the function myfunc() to the library my/lib. Note that it is necessary to specify the password when issuing a command which writes to a PDS. The (data) parameter tells PDS that the member is not an executable program

When a module contains more than one function it becomes necessary to use a PDS map file. The entry point specified in the map file is not important. The names of all the functions in the module which are to be accessible when searching must be placed on one line in the map file. For example, if a module file MYMOD/ASM contains three functions, func1, func2, and func3, then the map file, MYMOD/MAP, would look like this:

mymod/asm,func1,0,func2,0,func3,0

and the command to append the module to the library, MY/LIB:

pds(a) mymod my/lib (map)

PDS will assume that "mymod" has an extension of /MAP. The (map) parameter tells pds that MYMOD/MAP has the information needed to append the module.

Unlike in-line libraries, the order in which members are appended to a PDS library does not affect the functioning of the library. EDAS continues to search the library until no further references can be satisfied by the library. Thus, members may be appended in any order, purged and reappended, etc.

If you name your library, USER/LIB, LC will search your user library if you specify an **#option USERLIB** in your C-source programs after you insert the following lines into the LC/ASM file after ***GET CPROGRAM** .

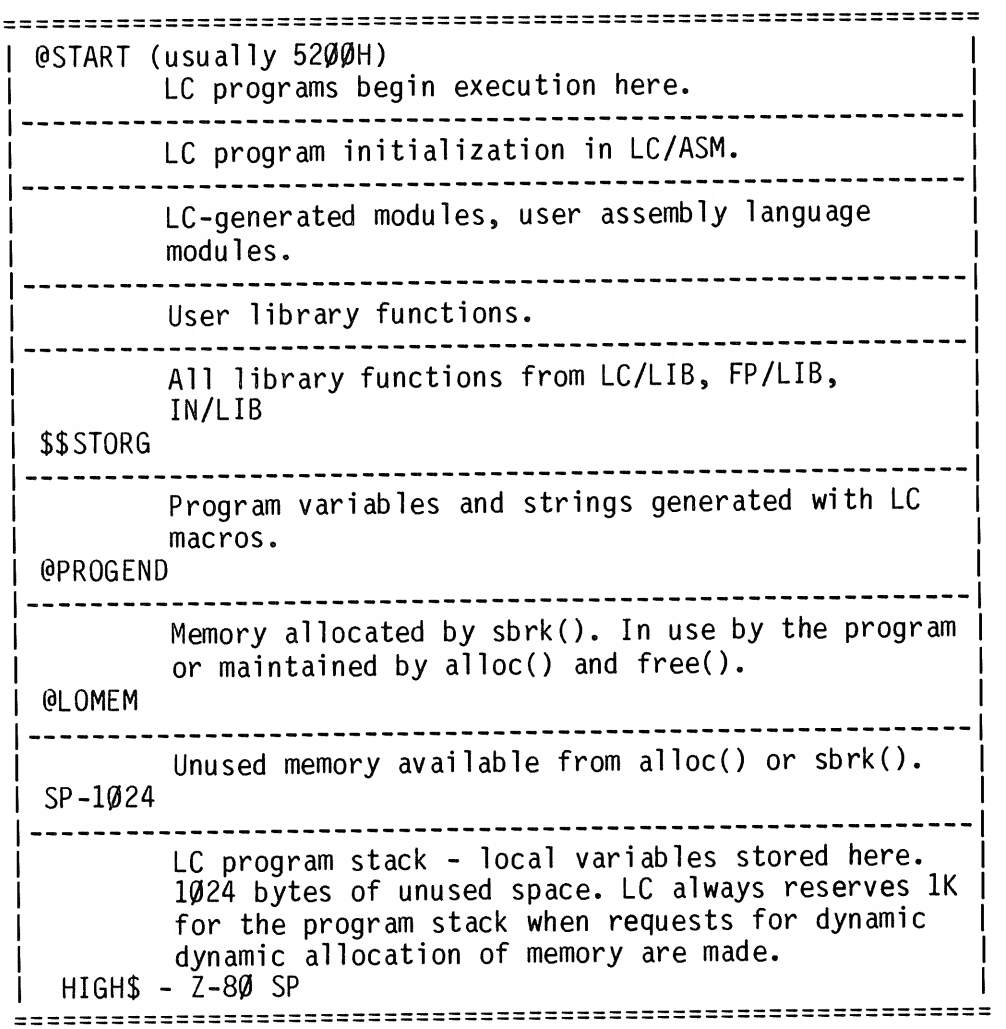
```
        IF      @_USERLIB
*SEARCH USER/LIB
        ENDIF
```

LC ASSEMBLY LANGUAGE OUTPUT STRUCTURE

LC generates a rather unusual assembly output file. LC depends on certain macro's in the file, LCMACS/ASM, which maintain and provide access to two separate program counters. These macros allow LC to declare variables and define strings in the middle of a function, without interrupting the actual generation of code to perform the function. The macros also cause all variables and strings to be placed together at the end of the program. These macros MUST BE USED if you are interfacing (with assembly language) to the variables generated in a module compiled by LC. Failure to access variables using the macros will result in the wrong address being used.

LC PROGRAM MEMORY MAP

LC programs, once assembled, have the following structure in memory:



LC MACROS

It is important to understand how to use macros in order to directly interface to variables declared by LC programs. However, it is not necessary if you are writing a function which will not access LC-declared variables. For more information on the use of the EDAS macro capability, see chapter six of the EDAS manual.

The macros in LCMACS/ASM are as follows:

$SORG

$SORG switches the EDAS program counter to the variable storage area. The current executable program counter is saved in $$TEMP, to be restored by a $PORG macro. Anything assembled by EDAS after $SORG will be placed at the end of the program, past all code assembled in the $PORG (normal) sections.

$PORG

This macro recovers the original program counter saved by the $SORG macro. The variable storage program counter is also saved, so that the next $SORG will continue where the last left off. WARNING: a $PORG must follow a $SORG only, not another $PORG. The proper program counter will be lost if a $PORG is done when not in the variable storage area.

$VAR #NAME,#SIZE

All external variables are declared by LC using the $VAR macro. A data area will be defined which is designated by the #NAME given, with the #SIZE given. Any references made to variables declared with the $VAR macro are made using the following macros:

$LSTR

This macro is used to prepare for the definition of a string. The HL register pair is loaded with the current storage program counter, and the EDAS program counter is switched to the storage area counter. The program counter must be in the program area before the $LSTR macro is used.

$LDS #NAME

This macro gets the address of the variable #NAME into the HL register pair.

$GETB #NAME

This macro fetches the contents of the character variable #NAME into the HL register pair. The H register will be loaded with zero, and L will contain the character.

$GETW #NAME

The $GETW macro fetches the two-byte integer stored at #NAME into the HL register.

$HS #NUM

The $HS macro points HL to the value "SP+NUM" (SP refers to the stack pointer). It is used to obtain the address of a local variable.

$PUTB #NAME

The $PUTB macro writes the character in the L register into character variable at #NAME.

$PUTW #NAME

The $PUTW macro will write the integer value in HL into the two-byte integer variable at #NAME.

LC IDENTIFIER OUTPUT

LC makes extensive use of the "*MOD" directive in EDAS. This EDAS assembler directive causes a unique one or two character alphabetic string to be assigned to a GET/SEARCH module. This replacement string is incremented each time another *MOD directive is encountered. Every occurance of '?' in a label is then replaced with the module's unique replacement string. For more information on the *MOD directive, see chapter five in your EDAS manual. The following table outlines the format of label generation from LC:

IDENTIFIER CLASS	LC OUTPUT
Temporary labels	$?#
External identifiers	NAME
-longer than 3 characters	NAME
-3 characters or less	NAM$
Static identifiers	
-external to functions	NAME@?
-internal to functions	NAME@?*
Goto labels	NAME$?*

NAME, NAM	LC identifier, 1 to 8 characters, upper case
#	The temporary label number
*	The function number (within the module)

LC generates labels in this fashion so that static variables, external variables, and labels will not conflict with each other. Thus, there can be an external variable named x, a module static named x in two different modules, a static named x in two different functions in the same module, and a (goto) label named x, all within the same program, with no conflicts. The '?' module substitution character in the labels will make labels unique within each module, while the function number appended to labels will make labels unique within each function. A dollar sign, '$', is appended to external labels which are three characters long or less. This prevents possible conflicts with register names and logical operators in EDAS.

Temporary labels are used by LC to implement conditional statements and operators, and loops. They are assigned numbers starting at 1, and incrementing by 1. The dollar sign, '$', as the first character allows you to suppress all temporary labels from your symbol table output by using the -SL switch of the EDAS <A>ssemble command. The '?' makes temporary labels unique within each module.

RUN-TIME SUBROUTINES

A program generated by LC performs logical, arithmetic, and data manipulation operations through two registers: the HL and DE register pairs. HL is the primary register; DE is the secondary register. The stack is used extensively to store intermediate results as expressions are evaluated. Certain operations are performed using subroutines in LC/LIB. These subroutines are:

SUBROUTINE	OPERATION PERFORMED
@GINT	get integer in memory at (HL) into HL
@PINT	put integer in HL into memory at (DE)
@OR	bitwise OR of DE with HL
@XOR	bitwise exclusive OR of DE with HL
@AND	bitwise AND of DE with HL
@EQ	returns DE == HL
@NE	returns DE != HL
@GT	returns DE > HL
@LT	returns DE < HL
@LE	returns DE <= HL
@GE	returns DE >= HL
@UGE	returns unsigned DE >= HL
@ULT	returns unsigned DE < HL
@UGT	returns unsigned DE > HL
@ULE	returns unsigned DE <= HL
@UCMP	unsigned compare DE-HL non-destructively
@ASR	arithmetic shift right of DE by HL
@ASL	arithmetic shift left of DE by HL
@NEG	returns two's complement of HL
@COM	returns one's complement of HL
@NOT	returns logical NOT of HL
@MULT	multiply DE by HL to HL
@DIV	divide DE by HL, returns remainder in DE, quotient in HL

ASSEMBLY LANGUAGE INTERFACING

While it is possible to insert assembly language source code directly into your LC program using the #asm-#endasm construct, it is much cleaner to interface by placing your assembly language code into a separate module. This keeps all the non-portable code separate from the portable LC code. It is best to call assembly language as a function, rather than including it directly into an LC function by mixing C and assembly source code.

REGISTER USAGE

All registers are available for use by the assembly language function. The only stipulation is that the stack pointer must be returned in the same condition as it was upon entry.

ARGUMENT PASSING

LC passes arguments on the Z-8Ø machine stack. Each argument is pushed onto the stack as a two-byte value. Arguments are pushed in order opposite of the order they are specified in the function call. Here is the assembly language which LC generates to perform a function call:

```
;       x=func(a,b,c);
        LD      HL,(C$)
        PUSH    HL
        LD      HL,(B$)
        PUSH    HL
        LD      HL,(A$)
        PUSH    HL
        CALL    FUNC
```

This process generates the following structure on the Z-8Ø machine stack:

```
(SP+6)  ==>     <c>
(SP+4)  ==>     <b>
(SP+2)  ==>     <a>
(SP+Ø)  ==>     return address
```

That is how the arguments appear to the called function when first entered. There are several methods which can be used within the called function to obtain the arguments. The simplest method is to POP each argument off the stack. This is not suitable for large numbers of arguments, but most efficient for 3 operands or less. Using the example above, the arguments could be retrieved as follows:

```
FUNC    POP     AF      ;return address saved
        POP     BC      ;argument <a> in BC
        POP     DE      ;argument <b> in DE
```

```
POP     HL      ;argument <c> in HL
PUSH    HL      ;restore argument <c>
PUSH    DE      ;  "        "     <b>
PUSH    BC      ;  "        "     <a>
PUSH    AF      ;stack is same as at entry
```

Note that the stack is returned to its original condition. It is always important to keep track of the stack pointer. However, the contents of the stack, i.e., the arguments, are "owned" by the called function and can be used like any local variable. A better method to use when dealing with large numbers of arguments is shown below:

```
LD      HL,2            ;offset to <a>
ADD     HL,SP           ;HL = address of <a>
CALL    @GINT           ;get contents of <a>
```

@GINT is a run-time library function which gets the integer pointed to by HL into HL. See the previous section on LC assembly output structure for more information.

Another method is to utilize the Z-8Ø index registers. The stack pointer must be placed into the index register first, then index offset values can be used to get and store the arguments:

```
LD      IX,Ø
ADD     IX,SP           ;get SP into IX
LD      L,(IX+2)        ;get LSbyte of a
LD      H,(IX+3)        ;get MSbyte of a
```

If an argument is intended to be a character variable, only the least significant byte (LSbyte) is needed, so a single indexed load is used.

LABELS AND CONSTANTS

It is strongly recommended that the assembly language programmer utilize the EDAS *MOD directive to assure that labels do not duplicate those in other modules. The following method will assure that you will not have this problem:

```
*MOD
VAR1@?  DW      Ø
EQU2@?  EQU     2
FUNC    ; your function here...
        . . .
        JR      FUNC2@?         ;go to temporary label
        . . .
FUNC2@?                         ;temporary label
```

By appending the @?, your label becomes unique from all others in an assembly, assuming that you placed a *MOD directive at the beginning of the module. Keep in mind, however, that other modules cannot get to these types of variables, since the '?' is substituted at assembly time. Labels which

must be accessed by other assembly language modules should be defined without the '?' in the label (i.e. to keep them global).

If a variable must be accessed by an LC module which is to be defined by the assembly module, then the macros described earlier in this section must be used to declare the variable, and to access it within the assembly language module. Examine the macros in LCMACS/ASM to see how they work, for additional interfacing ideas.

RETURNING A VALUE

LC programs use the HL register pair for a 16-bit accumulator. Any value to be returned by a called function must be placed in HL before returning to the calling function. Take care that a full 16-bit value is returned. If a character or 8-bit value is being returned, then H should be loaded with zero. If a true or false indication is to be returned, HL should be set to 1 or Ø, accordingly.

CALLING MACHINE LANGUAGE ROUTINES

The call() function has been provided in the installation library to standardize the invocation of machine language routines. The use of call() is documented on page 4-36 and illustrated in the program, syscall, which may be found in appendix D.

WHEN THINGS GO WRONG...

C is a language which offers great flexibility, but not without a price. The price of C's freedom is the programmer's ability to make catastrophic errors with ease. The programmer is not protected from himself when using LC. Your best protection is to carefully check your programs when you write them, for any evident errors before you try to run them. Of course, any time you test a program you should not have any disks in your drives that you would care about if they were suddenly erased. This is not to say that you shouldn't experiment; quite the contrary. However, always be prepared for the worst.

With LC, you have an advantage over other compilers. LC generates an assembly language source file. You can debug the program without second-guessing the compiler, or having to disassemble the compiled output. The modularity of the program also helps, since there are clear interfaces (functions) to breakpoint at. It would be helpful, though not essential, for the programmer to have familiarity with the Z-8Ø instruction set and with the debug facility of LDOS.

COMPILATION ERRORS

LC generates an error message whenever it finds something in the input file that cannot be recognized, or that doesn't fit the syntax of the C language. There are also some limitations in the LC implementation which can cause LC error messages to appear. When LC outputs an error message, it will print the line in error and point to the particular character where the error was recognized. The actual programming error is likely to be earlier in the program, depending on the type of error.

Some errors may not be detected until many lines later. For example, if a closing brace is missing in the input file, LC will not be able to detect the error until the next function declaration, which will then be flagged as a function call without an ending semicolon. This is because LC thinks the previous function has not been completed. Similarly, if an opening brace is missing, LC will not find out until the last closing brace is encountered, with no match.

In the appendix of this manual is a list of the error messages which LC can generate, and some likely causes for each. Most errors are usually typographical, but the user should be well versed in the C language and the LC implementation. Learn where to find information regarding syntax and capabilities of the language. The language definition chapter of this manual and the appendix of the K&R text are good places to look when not sure of syntax.

ASSEMBLY ERRORS

There are few assembly errors which can occur if you assemble your programs using LC/ASM and GET your compiled C modules. Your safest bet is to use LC/JCL when compiling and assembling. If you have written portions of your program in assembly language you may have a few more errors to deal with. Please read the previous section on assembly language interfacing for hints in debugging your assembly language.

If you are creating a CMD file directly from the LC/JCL file, you will be aware of an assembly error if the JCL aborts from EDAS. To specifically isolate the assembly error, you will need to execute EDAS, load the LC/ASM file, globally change CPROGRAM to the name of your C compiled output program, then assemble with the -WE switch. One of the following errors should prevail.

UNDEFINED SYMBOL - A symbol which you referenced in your program was undefined in any module in your program. This can be caused by omitting the definition, misspelling the identifier, or defining them incorrectly. A look at the name which is undefined will give you a clue as to which situation was the cause. Misspelling should be obvious. Look out for upper-case versus lower-case names. Remember that LC is case sensitive.

MULTIPLE DEFINITION - A symbol was defined twice with the same name. If you are assembling separately compiled or user library functions, you may have named two external variables or functions the same. To correct this, make one or both of the identifiers static within the module. If separate compilation is not being used, you have defined two external objects in your program with the same name.

It is useful to familiarize yourself with the previous section on LC assembly language output structure. Also, if you are assembling separately compiled or user library functions, a good understanding of the "extern" statement and external versus static variables is essential. Refer to the section on storage classes in Chapter 2, LC Language Definition.

APPENDIX - OPERATORS

UNARY OPERATORS

*	indirection (object at)	2-17
&	pointer (address of)	2-18
-	negation	2-18
!	logical NOT	2-18
~	one's complement	2-18
++	increment	2-18
--	decrement	2-18

BINARY OPERATORS

*	multiplication	2-21
/	division	2-21
%	modulus (remainder)	2-21
+	addition	2-21
-	subtraction	2-21
<<	shift left	2-21
>>	shift right	2-21
<	less than	2-21
>	greater than	2-21
<=	less than or equal to	2-21
>=	greater than or equal to	2-21
==	equal to	2-21
!=	not equal to	2-21
&	bitwise AND	2-21
^	bitwise exclusive OR	2-22
|	bitwise inclusive OR	2-22
&&	logical AND	2-22
||	logical OR	2-22
? :	conditional (if-then-else)	2-22
=	assignment	2-23
	[+=, -=, *=, /=, %=, <<=, >>=, &=, ^=, |=]	2-23

APPENDIX - OPERATORS

ptr = **alloc**(nbytes); 4-02
retcod = **atod**(str, dvar); 4-43
retcod = **atof**(str, fvar); 4-43
int = **atoi**(decs); 4-03
retcod = **box**(funcod, x1, y1, x2, y2); 4-25
retcod = **call**(address, regs); 4-36
retcod = **circle**(funcod, x1, y1, r1); 4-25
retcod = **cmd**("command string"); 4-35
cmdi("command string"); 4-35
retcod = **curpos**(); 4-35
cursor(col, row); 4-35
retcod = **dabs**(v1, v2); 4-42
retcod = **dadd**(v1, v2); 4-42
date(s); 4-35
retcod = **dcmp**(v1, v2); 4-42
retcod = **ddiv**(v1, v2); 4-42
retcod = **dfix**(v1, v2); 4-42
retcod = **dint**(v1, v2); 4-42
retcod = **dmul**(v1, v2); 4-42
retcod = **dsgn**(v1); 4-42
retcod = **dsub**(v1, v2); 4-42
retcod = **dtoa**(dvar, str); 4-43
intval = **dtoi**(fvar); 4-43
retcod = **dtof**(dvar, fvar); 4-43
exit(code); 4-04
retcod = **fabs**(v1, v2); 4-40
retcod = **fadd**(v1, v2); 4-40
retcod = **fatn**(v1, v2); 4-40
retcod = **fclose**(fp); 4-05
retcod = **fcmp**(v1, v2); 4-40
retcod = **fcos**(v1, v2); 4-40
retcod = **fdiv**(v1, v2); 4-40
retcod = **fexp**(v1, v2); 4-40
retcod = **ffix**(v1, v2); 4-40
eofind = **fgets**(buf, max, fd); 4-06
fill(buffer, count, char); 4-34
retcod = **fint**(v1, v2); 4-40
retcod = **flog**(v1, v2); 4-40
retcod = **fmul**(v1, v2); 4-40
fp = **fopen**(fspec, mode); 4-07
fpinit(); 4-39
retcod = **fraise**(v1, v2); 4-40
retcod = **fprintf**(fp, control, arg1, arg2, ...); 4-08
retcod = **fputs**(string, fp); 4-09
free(ptr); 4-10
retcod = **freemem**(); 4-34
retcod = **frnd**(v1, v2); 4-40
retcod = **fsgn**(v1); 4-40
retcod = **fsin**(v1, v2); 4-40
retcod = **fsqr**(v1, v2); 4-40
retcod = **fsub**(v1, v2); 4-40
retcod = **ftan**(v1, v2); 4-40
retcod = **ftoa**(fvar, str); 4-43
retcod = **ftod**(fvar, dvar); 4-43

APPENDIX - OPERATORS

intval = **ftoi**(fvar); 4-43
c = **getc**(fp); 4-11
c = **getchar**(); 4-12
eofind = **gets**(buffer); 4-13
retcod = **inkey**(); 4-34
retcod = **inport**(port); 4-34
retcod = **isalpha**(char); 4-15
retcod = **isdigit**(char); 4-15
retcod = **islower**(char); 4-15
retcod = **isupper**(char); 4-15
itoa(int, decs); 4-Ø3
retcod = **itod**(ivar, dvar); 4-43
retcod = **itof**(ivar, fvar); 4-43
itox(int, hexs); 4-Ø3
retcod = **line**(funcod, x1, y1, x2, y2); 4-25
move(pfrom, pto, len); 4-14
outport(port, value); 4-35
retcod = **pixel**(funcod, x, y); 4-23
ploc(address); 4-27
retcod = **pmode**(quadrant); 4-27
retcod = **point**(x, y); 4-23
printf(control, arg1, arg2, ...); 4-16
cret = **putc**(c, fd); 4-17
cret = **putchar**(c); 4-18
retcod = **puts**(string); 4-19
reset(x, y); 4-23
ptr = **sbrk**(nbytes); 4-2Ø
set(x, y); 4-23
strcat(dest, source); 4-3Ø
retcod = **strcmp**(source, dest); 4-3Ø
strcpy(dest, source); 4-31
retcod = **strepl**(source, dest, pos, count); 4-31
strept(dest, source, repeat); 4-31
retcod = **strfind**(dest, source, pos); 4-32
stright(dest, source, count); 4-32
strleft(dest, source, count); 4-32
retcod = **strlen**(source); 4-32
retcod = **strmid**(dest, source, pos, count); 4-32
time(s); 4-35
c = **tolower**(char); 4-21
c = **toupper**(char); 4-21
int = **xtoi**(hexs); 4-Ø3

APPENDIX - ERROR MESSAGES

LC ERROR MESSAGES

64> "already defined" - The object being declared has already been defined in the module.

65> "argument error" - The expression just processed caused LC to input the next line before the function call was completely processed. See the "split function call" error for additional information.

66> "bad label" - The label specified in a "goto" statement was not a valid LC identifier.

67> "expected comma" - The context of the input required a comma, but none was found.

68> "function not declared" - The context of the input demanded that a function be declared, i.e., the input did not match anything which could be a compiler directive or a variable declaration, so it was assumed that a function was being declared.

69> "global symbol table overflow" - LC ran out of global symbol table space. Either decrease the amount of memory reserved in high memory (see "not enough memory" error) or split the module being compiled into smaller modules with fewer external variables.

70> "illegal address" - the "&" (address of) operator was used with an expression which was not an object in memory.

71> "illegal argument name" - The argument name is not a valid LC identifier.

72> "illegal identifier" - The input was interpreted as an identifier, but did not conform to LC's rules for identifiers.

73> "input file open failure" - An input file could not be opened successfully.

74> "invalid expression" - The input could not be recognized as an expression when the context of the program required an expression.

75> "invalid option name" - The name given in a "#option" compiler directive was not a valid LC identifier.

76> "invalid option value" - The value given in the "#option" compiler directive was not a character literal or numeric constant.

77> "local symbol table overflow" - LC ran out of local symbol table space. Either decrease the amount of memory reserved in high memory (see "not enough memory" error) or decrease the number of local variables in the function.

78> "line too long" - The input line exceeded the maximum input line size allowed by LC (128 characters/line).

APPENDIX - ERROR MESSAGES

79> "macro table full" - The current "#define" caused the table space allocated by LC to be exceeded. Either decrease the amount of memory reserved in high memory (see "not enough memory" error) or decrease the number of "#define" statements in the module.

8Ø> "missing ':'" - A "?" operator was found without a matching ":" operator.

81> "missing apostrophe" - A character literal was not ended with an apostrophe. LC does not allow a character literal to be continued on another line

82> "missing bracket" - The ending "bracket" character shown in the error message was expected, but was not found.

83> "missing quote" - A string literal did not have an ending quote ('"'). Strings cannot be continued on another line.

84> "missing semicolon" - No semicolon was found at the end of a statement. The ';' character is the statement terminator, and must be placed at the end of a simple statement.

85> "must be constant" - The size of an array must be a numeric constant.

86> "must be extern" - a function declaration was encountered past a comma in an external (but not "extern") declaration statement. This context implies that the function is of storage class, "extern", but this was not the class of the declaration statement.

87> "must be lvalue" - The expression being processed specifies that a value be placed into an object, but no object which could be stored into was found.

88> "negative size illegal" - An array with a negative size was declared. The array size is made positive before being used.

89> "nested too deep - ignored" - The #include statement would have nested too deeply, if not ignored. Up to eight (8) nesting levels are available in LC.

9Ø> "no closing brace" - The end of the last input file was encountered without a closing brace for the current function being found.

91> "no input file" - No input files were specified on the command line which invoked LC.

92> "no multiple dimensions" - The array being declared has more than one dimension, which is not supported. LC supports one-dimensional arrays only.

93> "no while after do" - A "do" statement was compiled, but no "while" statement followed it.

94> "not a pointer expression" - the indirection operator, "*", was used with an expression which does not result in the address of an object in memory.

APPENDIX - ERROR MESSAGES

95> "not enough memory" - When LC began execution, not enough memory was free for LC to execute properly. Decrease the amount of modules in high memory (filters, MINIDOS, KSM, SYSRES'ed overlays, etc.) and try to execute LC again.

96> "output file error" - An error occurred while writing to the output file.

97> "output file open error" - An error occurred when attempting to open the LC output file.

98> "split 'for'" - The expressions in parentheses in a for statement must appear on the same line. This is a limitation of the LC implementation, not a limitation of the C language.

99> "split function call" - All the arguments in a function call must be given on the same line. This is a limitation of the LC implementation, not a limitation of the C language.

100> "too indirect" - The expression exceeded the amount of indirection allowed by the declaration of the objects used in the expression.

101> "too many active loops" - LC allows nesting of loops and "switch" statements to 25 levels. The loop or "switch" being processed would have nested more than 25 levels.

102> "too many arguments" - There were too many arguments specified in a function call. LC limits the number of arguments in a function call to 16. This is a limitation of the LC implementation, not a limitation of the C language.

103> "unmatched arguments" - The argument being declared did not match any of those in the argument list for the function.

104> "unrecognizable declaration" - The object being declared contains a character which is invalid in the context of a declaration.

105> "unrecognizable option" - One of the options specified in the command line was not a valid LC option.

APPENDIX - SAMPLE PROGRAMS

```
/* CAT/CCC */
/*  Utility to concatenate files to standard output  */
/*   adapted from "The C Programming Language" by    */
/*   Kernighan and Ritchie.                          */
#include stdio/csh
FILE *fp;
main(argc,argv)
    int argc,*argv[];
{
    if (argc == 1) /* no args, copy standard input */
         filecopy(stdin);
    else
         while (--argc)
         {    if ((fp = fopen(*++argv,"r")) == NULL)
              {    fputs(*argv);
                   abort(" - open error.");
              }
              else
              {    filecopy(fp);
                   fclose(fp);
              }
         }
}
filecopy(fp)    /* copy a file to the standard output */
    FILE *fp;
{
    int c;
    while ((c = getc(fp)) != EOF)
         if (c != putc(c,stdout))
              abort("Output file write error");
}
abort(msg)
    char *msg;
{
    fputs(msg,stderr);
    putc(eol,stderr);
    exit(-1);    /* let system know about the error */
}
```

```
/* Test of cmdi() */
#include stdio/csh
#option INLIB
char buf[100];
main()
{   puts("Test of cmdi()\n\n");
    puts("Enter command:");
    gets(buf);
    cmdi(buf);
}
```

```
/* Test of cmd() */
#include stdio/csh
#option INLIB
char buf[1ØØ];
main()
{   int rc;
    puts("Test of cmd()\n\n");
    puts("Hit break to exit back to DOS\n");
    while (TRUE)
    {   puts("Enter command:");
        if (gets(buf)==NULL)
            exit(Ø);
        rc=cmd(buf);
        printf("\nReturn code is %d\n",rc);
    }
}
```

```
/* compare/ccc - 03/08/83 */
#include stdio/csh
#option INLIB
int line, c, c1, c2, record, bytes;
FILE *fp1,*fp2;
main (argc,argv)
    int argc, *argv;
{   if (argc!=3)
    {   puts("Format error:  compare file1 file2\n");
        exit();
    }
    line = 1; bytes = -1; record = 0;
    fp1 = getfile(*++argv);
    fp2 = getfile(*++argv);
    while ((c1 = getc(fp1)) != EOF  &
            (c2 = getc(fp2)) != EOF)
   {   ++bytes;
       if (bytes == 256)
           { ++record; bytes = 0; }
       if (c1 != c2)
       {printf  ("Difference: line %-8d,  record  X'%2.2x', byte  X'%2.2x'\n",
line,record,bytes);
       printf("     File1: <X'%2.2x'>  File2: <X'%2.2x'>\n",c1,c2);
       if (c = inkey() != 0 ) exit(-1);}
        else if (c1 == eol) ++line;
    }
    if (c1 == c2)
        puts ("The files are of equal length.\n");
    else
        puts ("The files are not of equal length\n");
}
getfile(fname)
    char *fname;
{   char *fp;
    if ((fp=fopen(fname,"r")) == NULL)
    {   printf("Open error - %-20s\n",fname);
        exit();
    }
    else return fp;
}
```

```
/* decom/ccc */
/*  comment stripper program - 10/01/83 */
#define eof -1
#define eol 13
int col, tab, string, comment;
int c;
main()
{  col = tab = comment = string = 0;
   while ( (c=getchar() )!=eof)
   {   if (comment)
       {   if (c==eol) tab = comment = string = 0;
           else continue;
       }
       else if (c==';')
           {
           if (!string)
               {   comment = 1; continue; }
           }
       else if (c==eol)
           tab=string=0;
       else if (c=='\'')
           string = !string;
       if (whitespace(c) && !string)
           { tab = 1; continue; }
       else if ( tab )
           { putchar('\t'); tab = 0; col++; }
       if (c == eol )
           { if ( col == 0 )
               continue;
           else col = -1;
           }
       putchar(c); col++;
   }
   exit(0);
}
whitespace(c)
   int c;
{
   if (c == '\t' || c == ' ')
       return(1);
   else
       return(0);
}
```

```
/* Plot Hilbert curves of orders 1 to n */
#include stdio/csh
#option INLIB
int h,x,y,xØ,yØ,u,v;
main()
{   int i,n, hØ;
   puts("\xØf\x1c\x1fhilbert curves\n");
    n = 4; hØ = 32;
   i=Ø; h=hØ; xØ=h / 2; yØ=xØ;
   while ( i < n )
         {
         h = h / 2;
         xØ = xØ + h / 2; yØ = yØ + h / 2;
         x = xØ+ i * 32; y = yØ+1Ø; u=x; v=y;
         ++i; p(1,i);
         }
    exit(Ø);
}
move()
    {   int i,j;
    for (i=min(x,u) ; i < (max(x,u)+1) ; i++)
      for (j=min(y,v) ; j < (max(y,v)+1) ; j++)
         pixel(1,i,j);
    u=x; v=y;
    return(Ø);   }
min(a,b)
    int a,b;
    {   if (a > b) return(b); else return(a);   }
max(a,b)
    int a,b;
    {   if (a < b) return(b); else return(a);   }
p(typ,i)
    int typ, i;
    {    if (i <= Ø) return(Ø); else
         switch (typ) {
         case 1: p(4,i-1); x = x-h; move();
            p(1,i-1); y = y-h; move();
            p(1,i-1); x = x+h; move();
            p(2,i-1) ; break;
         case 2: p(3,i-1); y = y+h; move();
            p(2,i-1); x = x+h; move();
            p(2,i-1); y = y-h; move();
            p(1,i-1) ; break;
         case 3: p(2,i-1); x = x+h; move();
            p(3,i-1); y = y+h; move();
            p(3,i-1); x = x-h; move();
            p(4,i-1) ; break;
         case 4: p(1,i-1); y = y-h; move();
            p(4,i-1); x = x-h; move();
            p(4,i-1); y = y+h; move();
            p(3,i-1); break;
         }
         return(Ø);
       }
```

```
/* linetest/ccc */
#include stdio/csh
#option INLIB
main()
{
   int x1,x2,y1,y2,t,t1;
   puts("\x1c\x1f"); /* clear the screen */
   for ( x1=Ø, y1=Ø, x2=127, t = Ø ; t <= 47 ; t++ )
      {  line(1,x1,y1,x2,t);
         line(Ø,x1,y1,x2,t);
      }
   for ( y2=47,t = 127 ; t >= Ø ; t-- )
      {  line(1,x1,y1,t,y2);
         line(Ø,x1,y1,t,y2);
       }
   exit(Ø);
}
```

```
/* xfer/ccc */
#include stdio/csh  /* standard I/O definitions */
/*   XFER - copy standard input to standard output   */
int  c, bytes, lines;
FILE *fp;
main()
{
    bytes = lines = Ø;
    while( (c=getchar()) != EOF)
    {    putchar(c);
         ++bytes;
         if (c == EOL) ++lines;
    }
    fp = fopen("*do","w");
    fprintf(fp, "%d characters , %d lines", bytes, lines);
}
```

A P P E N D I X - S A M P L E P R O G R A M S

The following example program may help to illustrate the use of CALL. The program was provided by Rich Deglin.

```
/* test of syscall() */
/*  Use.......Model I    Model III   LDOS/TRSDOS 6.x */
#define CKDRV 0x44B8      0x4209        33
#define DATE  0x4470      0x3033        18
#define DODIR 0x4463      0x4419        34
#define DSPLY 0x4467      0x4467        10
#define TIME  0x446D      0x3036        19
#define AF 0
#define BC 1
#define DE 2
#define HL 3
#define IX 4
#define IY 5
#define CARRY 0
#option INLIB
int rc,d;
char *regs[6],buf[100];
main()
{   puts("Test of call()\n\n");
    for (d=0;d<8;++d)
    {   regs[BC]=d;
        rc=call(CKDRV,regs);
        printf("Drive %d %s%s\n",d,rc?"not ":"","ready");
        if (regs[AF]&(1<<CARRY))
            puts("Drive is write protected\n");  }
    wait();
    dt(DATE,"Date");
    dt(TIME,"Time");
    regs[HL]="This is a message\n";
    call(DSPLY,regs);
    wait();
    regs[BC]=(4<<8)+0; regs[HL]=buf;
    call(DODIR,regs);
    strmid(buf+80,buf,0,8);
    strmid(buf+90,buf,8,8);
    printf("%s %s free: %dK\n\n",buf+80,buf+90,buf[18]+(buf[19]<<8));
    regs[BC]=0;
    call(DODIR,regs);
    wait();
}
wait()
{   puts("\nHit any key"); getchar(); clscrn(); }
dt(addr,str)
    char *addr,*str;
{   regs[HL]=buf;
    call(addr,regs);
    *regs[HL]=0;
    printf("%s: %s\n",str,buf);
}
clscrn()
{   puts("\x1c\x1f");  }
```